L'intelligence artificielle dans les jeux

Anestakis Petros, Cortild Daniel,
Fischer Emile, Janusis Eimantas,
Olimid Dominic

Collège Saint-Michel d'Etterbeek, 1040 Bruxelles

CSSMTicTacToe@gmail.com

Avril 2020

Table des matières

1 Introduction

"All models are wrong. But some are usefull".

Cette phrase célèbre du statisticien anglais George Box reflète les défis que rencontre la Science en général et en particulier face à l'apprentissage automatique. Depuis la nuit des temps, l'Homme essaie de comprendre tout ce qui l'entoure en construisant des modèles simplifiés de celui-ci. Ces derniers décrivent donc une réalité imparfaite, et n'en reflète qu'une partie infime. L'erreur d'un modèle varie et dépend de l'objectif à atteindre en faisant ce modèle, mais elle sera toujours présente même si il existe une multitude de processus pour la réduire au maximum.
Grâce à tous ces procédés, certains de ces modèles arrivent à analyser des situations réelles avec une erreur acceptable ce qui peut les rendre incroyablement utiles.

Prenons pour exemple les cartes. Elles sont utilisées dans la vie de tous les jours pour un grand nombre de personnes et illustre bien nos propos. Qu'elles soient électroniques ou physiques, celles ci ont pour but de modéliser l'emplacement des objets les uns par rapport aux autres avec le plus de précision possible, mais même si ces cartes deviennent de plus en plus précises de nos jours, elles ne seront jamais parfaitement représentatives de la réalité, pourtant personne n'oserait prétendre qu'elles ne soient utiles!

Comme tout processus, un jeu [1] peut être modélisé d'un point de vue mathématique. De plus, on peut s'intéresser à l'optimisation de ces modèles: cette étude est appelée la théorie des jeux. Dans ce travail nous discuterons de quelques approches générales utilisées dans l'apprentissage automatique en lien avec la théorie des jeux.

Nous allons créer tout au long du travail plusieurs modèle mathématiques capables de jouer à divers jeux et d'y gagner. Nous appelons ces modèles des modèles mathématiques, mais ils n'auraient pas lieu sans les développements énormes en informatique de ces 50 dernières années. En effet, ces modèles mathématiques sont compliqués au point de ne pas être exécutables, voire même interprétables, par les êtres humains.

Nous allons commencer, à la section 2, par un algorithme simple. On l'appelle simple car il est réellement simple, et même un humain s'y retrouverait facilement. Nous allons adapter cette méthode à un problème connu, étant le problème de passage de rivière.

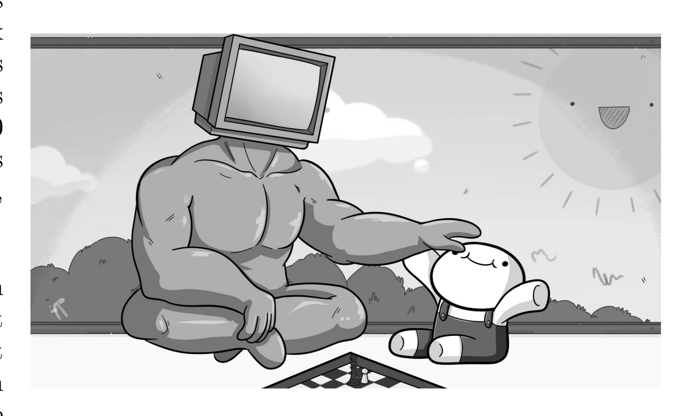
À la section 3, nous allons développer un algorithme "brute-force", dans le sens où il est très brutal. En effet, l'algorithme Min-Max cherche toutes les combinaisons possibles dans le jeu pour vérifier si la position est gagnant ou pas. Cela s'adapte très facilement dans des jeux simples, mais beaucoup moins bien lorsqu'ils deviennent plus complexes. Le jeu que nous avons choisi pour illustrer cette méthode est le jeu de morpion, également connu sous le nom de OXO.

La section d'après va exploiter la base des stratégies algorithmiques en théorie des jeux; l'apprentissage par renforcement. Il va se baser sur un système de récompense donnée au joueur. Dans cette partie, nous allons utiliser cet algorithme pour se déplacer sur une grille 4×4.

La section 5 va se focaliser sur le Q-Learning, avec une illustration dans un jeu où un taxi doit se déplacer et ramasser/déposer des personnes.

Les deux sections qui suivent vont servir d'introduction au Deep Learning, une méthode d'apprentissage proche de celle des humains. La section 6 va introduire le concept dans un contexte non-lié aux jeux, bien qu'il ne soit pas directement lié aux jeux. On fournira un exemple de reconnaissance de chiffres écrits à la main. La section 7 est une combinaison entre le Deep Learning et le Q-Learning, formant le Deep Q Learning, que nous allons adapter au jeu d'échecs.

Toutes ces méthodes, sauf la première, ne sont pratiquement pas applicables par des êtres humains. Pourtant, aucune ne pourrait exister sans l'intelligence humaine. Dès lors, il n'y a pas besoin d'avoir peur que ce genre d'intelligences artificielles vont dominer le monde un jour.



[1] Un jeu mathématique est un jeu dans lequel les règles, stratégies et résultats sont dictés par des paramètres mathématiques clairs

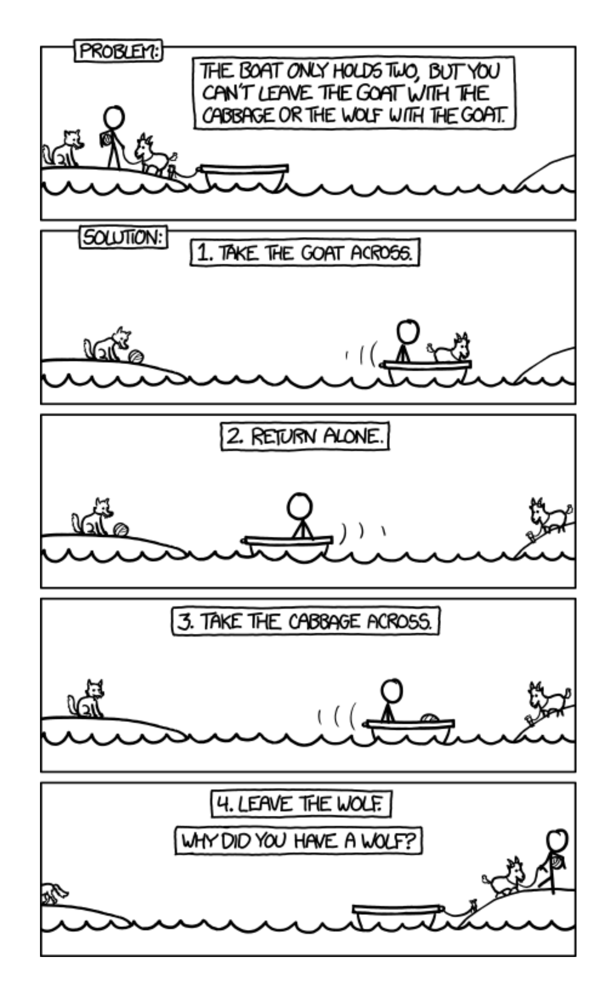
2 Algorithmes Simples

2.1 Loup, Chèvre et Chou

Un jour, un berger est allé au marché pour acheter diverses choses. Finalement, il retourne chez lui avec un loup, une chèvre et un énorme chou. Pour revenir à sa maison, il doit traverser un lac. Le seul problème est que le bateau pour traverser ne contient que deux places, et que le berger, le loup, la chèvre et le chou occupent chacun une place entière. Il ne peut laisser la chèvre seule avec le chou, comme quoi elle le mangerait, ni le loup seul avec la chèvre, comme quoi il la mangerait. Comment doit-il procéder pour réussir à tout ramener chez lui?



Nous vous encourageons à prendre 4 bouts de papier et d'y inscrire les noms respectifs des objets en jeu durant ce problème et d'essayer par vous même. Bien que le problème ne soit pas impossible, il risque de poser certains problèmes.



Pourtant ce problème possède une solution très simple, que l'on appellera un algorithme.

Notre algorithme ressemble à ceci:

• Le berger prend la chèvre avec de l'autre côté, laissant le chou avec le loup.
• Le berger revient seul.
• Le berger prend le chou, laissant seul le loup du côté du marché.
• Le berger revient avec la chèvre, laissant du côté de sa maison le chou seul.
• Le berger laisse la chèvre du côté du marché et revient avec le loup.
• Le berger revient seul, laissant du côté de sa maison le loup et le chou seuls.
• Le berger prend la chèvre et retourne de l'autre côté, pour avoir ses 3 achats du côté de sa maison et pouvoir revenir chez lui.

Cet algorithme sera appelé un algorithme simple, même s'il n'est pas si simple à trouver. Pourtant, après l'avoir vu, tout humain sachant lire saura résoudre le problème. C'est pour cela que nous l'appellerons simple.

2.2 Problème du Détachement

Un autre exemple est le problème du détachement. Un détachement de soldats arrivent devant un pont cassé, et le courant est trop fort pour pouvoir traverser la rivière à la nage. Ils trouvent alors un bateau opéré par 2 enfants. Il est petit, et ne sait contenir qu'un seul soldats, ou au plus 2 enfants. Pourtant, le capitaine trouve un moyen pour ramener toute sa troupe de l'autre côté, tout en rendant le bateau aux enfants. Comment fait-il?

Nous ne savons pas combien de soldats comporte la troupe, donc cela nous donne l'idée de devoir faire quelquechose de récursif. Au départ, les deux enfants sont du côté des soldats. Nous voulons donc que les deux enfants aident un soldats à la fois de passer, tout en revenant tous les deux du côté initial des soldats après en avoir fait passer un.

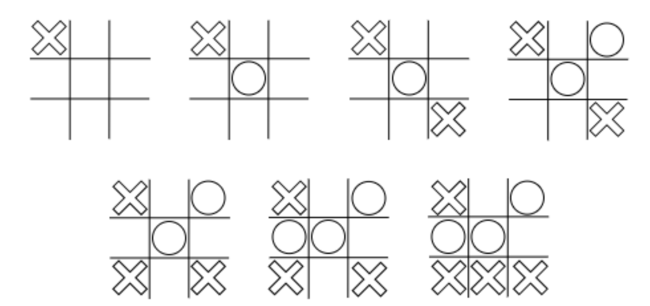
C'est à cela que ressemblera notre algorithme simple:

Tant qu'il reste des soldats à faire traverser:
• Les deux enfants traversent la rivière.
• Un des deux enfants revient du côté du soldat qui veut traverser.
• Le soldat traverse seul.
• L'autre enfant reprend le bateau pour revenir.

De nouveau, bien que la solution ne soit pas triviale, elle est très facile à appliquer. En général, tout algorithme suivant des ordres d'une liste seront appelés algorithmes simples.

3 Algorithme MinMax

Nous allons appliquer l'algorithme MinMax sur le jeu connu de morpion, mais il se généralise aisément. Ce jeu est très simple; Deux joueurs jouent à tour de rôle dans une grille 3×3, en plaçant leur symbole ("X" pour le joueur 1 et "O" pour le joueur 2) dans une case vide. Le premier qui aligne 3 de ses symboles remporte la victoire. Il y a possibilité de match nul.



3.1 Graphes, arbres et racines

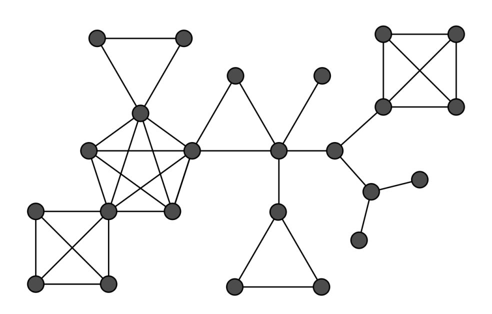
En mathématiques, et plus spécialement en théorie des graphes, on appelle un *graphe* une structure composée d'objets (nommés *sommets* ou *noeuds*) et de relations entre ces objets (nommés *arrêtes* ou *liens*).
On dénote le graphe composé des sommets V ("V" pour **V**ertices en anglais) et des relations E ("E" pour **E**dges en anglais) par

$$\mathcal{G}(V, E)$$

Lorsque le contexte est clair, on omettra parfois les paramètres et on parlera du graphe comme étant le graphe $\mathcal{G}$.
L'ensemble E est un sous-ensemble des paires non-ordonnées de V, ce qui est noté

$$E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$$

Voici un exemple de graphe:



On appelle $(u_0, u_1, u_2, \ldots, u_k)$ un *chemin* de $\mathcal{G}$ si et seulement si

$$\forall i \in [0, k-1], \quad \{u_i, u_{i+1}\} \in E$$

On dira qu'un graphe $\mathcal{G}$ est *connexe* si et seulement si

$$\forall u, v \in V, \quad \exists u_1, u_2, \ldots, u_k \text{ tel que}$$
$$(u, u_1, u_2, \ldots, u_k, v) \text{ est un chemin}$$

Ceci revient à ce que le graphe $\mathcal{G}$ soit connexe si et seulement si il ne peut Être partagé en 2 graphes disjoints.

Un graphe sera appelé *acyclique* si

$$\forall u_0 \in V, \quad \nexists u_1, \ldots, u_k \in V$$
$$\text{tel que } \big(\forall i \neq j \in [0, k],\ u_i \neq u_j\big) \ \wedge$$
$$\big(\forall i \in [0, k-1],\ \{u_i, u_{i+1}\} \in E\big)$$

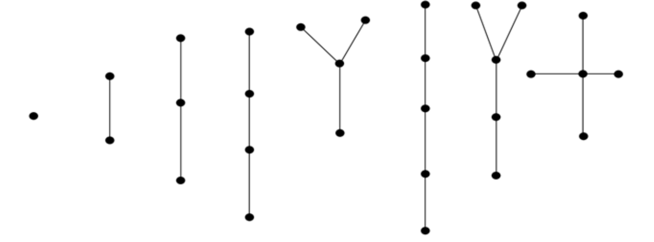
Ceci revient à ce que le graphe n'ait aucun chemin revenant sur soi-même, qu'à aucun moment il ne crée un cycle, une boucle.

Tous les graphes mentionnés jusqu'à présent sont considérés *non-orientés*. Cela veut dire que les arrêtes ne sont pas dotés d'un sens, donc

$$\{u, v\} \in E \iff \{v, u\} \in E$$

Nous pouvons dès lors définir un arbre:
Un *arbre* est un graphe connexe non-orienté et acyclique.

Voici quelques exemples d'arbres:



Un graphe sera appelé *orienté* si les arrêtes sont dotés d'un sens, ce qui veut dire qu'une arrête a un point de départ et un point d'arrivée, sans que les deux puissent jouer les deux rôles. En effet, dans un graphe orienté,

$$(u, v) \in E \implies (v, u) \notin E$$

Dans un graphe orienté, nous allons définir l'ensemble $e(X)$ pour un noeud $X \in V$ par

$$e(X) = \{u \mid (X, u) \in E\}$$

Cet ensemble $e(X)$ sera communément appelé les *enfants* de X.

De façon analogue, on définit les *parents* d'un noeud X comme

$$p(X) = e^{-1}(X) = \{u \mid (u, X) \in E\}$$

Nous allons maintenant considérer des arbres enracinés. Un tel graphe est juste un arbre orienté

tel que chaque noeud, sauf un, appelé la *racine* de l'arbre, ait exactement un parent.

Le "sauf un" est important et ne peut être négligé dans la définition de l'arbre enraciné. En effet, montrons que si elle est retirée des définitions, alors on retombe sur une contradiction avec les autres définitions.

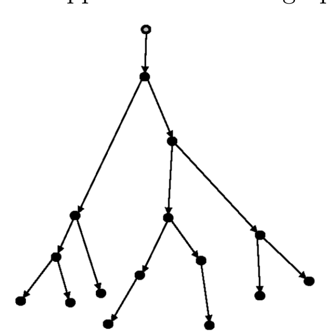
Propriété: Soit $\mathcal{G}(V, E)$ un graphe connexe orienté. Si tout noeud $u \in V$ a au moins un parent, alors $\mathcal{G}$ possède un cycle.

Preuve Propriété: Supposons que $\mathcal{G}$ respecte les hypothèses et qu'il soit acyclique. Soit $(u_0, u_1, \ldots, u_k)$ un chemin de longueur maximale. Comme $\mathcal{G}$ ne possède pas de cycles, k est fini et on peut donc dire que $k+1 > k$. Par hypothèse, il existe $v \in V$ tel que $(v, u_0) \in E$. Mais alors $(v, u_0, u_1, \ldots, u_k)$ est un chemin de $\mathcal{G}$, de longueur $k+1 > k$. Ceci est absurde comme on avait pris le chemin de longueur maximale. Il n'existe donc pas de chemin de longueur maximale, et comme V est fini, $\mathcal{G}$ doit forcément posséder un cycle. □

Dans la définition d'un graphe enraciné, on suppose qu'il n'existe qu'un seul noeud $r \in V$ tel que

$$(v, r) \notin E \quad \forall v \in V$$

Ce noeud r sera appelé la racine du graphe.



On dénote par $l(u)$ les feuilles (**L**eaf en anglais) de $\mathcal{G}$, c'est à dire les noeuds n'ayant pas d'enfants;

$$l(u) = \{u \mid e(u) = \emptyset\}$$

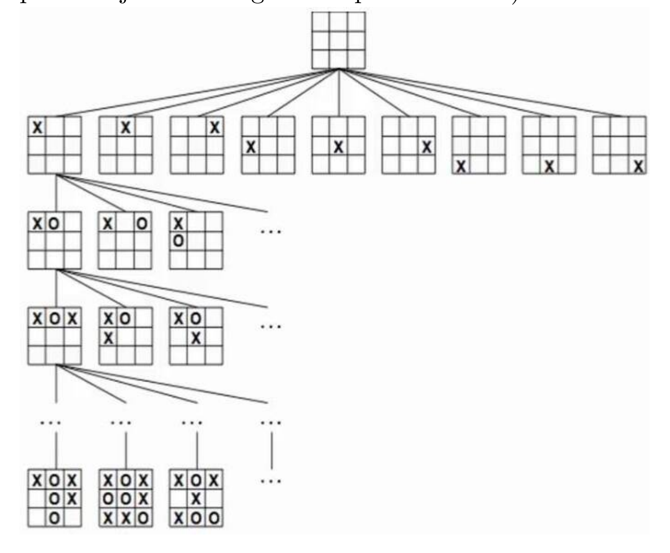
3.2 Arbre de jeu

Nous allons représenter un jeu par l'intermédiaire d'un arbre de jeu. Un arbre de jeu est un arbre enraciné dont chaque noeud représente un état de jeu. La racine correspondra à l'état initial du jeu, et les enfants d'un noeud correspondent aux états accessibles à partir de ce noeud en une seule action. Dans le jeu d'OXO, la racine représentera la grille vide.
On appelle profondeur d'un noeud la distance du chemin le plus court vers la racine. En d'autres mots, c'est le nombre d'actions requises pour arriver à cet état.

Pour deux états de jeu u et v on notera

$$(u \to v) \iff ((u, v) \in E)$$

La profondeur maximale de l'arbre du OXO sera de 9, comme le jeu se finit obligatoire après 9 actions. Comme c'est le joueur "X" qui commence, les enfants de la racine seront au nombre de 9 et représenteront chacune des possibilités de jeu du joueur "X". Chacun de ces noeuds de profondeur 1 aura 8 enfants, chacun représentant une possibilité de jeu du joueur "O" dans la grille contenant déjà un "X". On continue ainsi jusqu'à être arrivé à une position finale (Soit gagnante pour un joueur ou égalitaire pour les deux).



Il faut faire au moins 5 actions pour pouvoir finir le jeu, et le jeu peut durer au plus 9 actions. Un jeu dans lequel aucun joueur n'a gagné dure obligatoirement 9 actions. Après avoir joué quelques parties, on se rend compte que l'égalité arrive assez souvent, donc on va supposer grossièrement que tout jeu dure 9 actions.

À la profondeur 0, tous les noeuds représentent des états comportant aucun signe, donc ayant $9 = 9 - 0$ cases libres. À la profondeur i, les noeuds représentent des états contenant une case libre de moins qu'à la profondeur $i-1$. On peut ainsi montrer par récurrence que les noeuds de la profondeur i représentent des états contenant $9-i$ cases libres.
Ainsi, un noeud de profondeur i à exactement $9-i$ enfants, comme chaque enfant est simplement le noeud parent ajouté d'un signe dans une case libre. Comme chaque noeud de même profondeur a le même nombre d'enfants et comme le nombre de descendants de profondeur $i+1$ d'un noeud de profondeur i est exactement $(9-i)$, on en déduit que le nombre de descendants de profondeur j d'un noeud de profondeur $i < j$ est de

$$(9-i) \cdot (9-(i-1)) \cdots (9-j+1)$$

Nous pouvons dès lors déduire que le nombre de descendants de profondeur i du noeud initial, dénoté par D_i, est de

$$D_i = 9 \cdot 8 \cdots (9-i+1) = \frac{9!}{(9-i)!}$$

Le ! est définie comme suit:

$$!: \mathbb{N} \to \mathbb{N}, x \mapsto x \cdot (x-1) \cdot (x-2) \cdot \dots \cdot 2 \cdot 1$$

Par convention, on prend $0! = 1$.
Ainsi, le nombre total de noeuds dans l'arbre est

$$\sum_{i=0}^{9} D_i = 9! \sum_{i=0}^{9} \frac{1}{i!} = 623530$$

Ce nombre reste totalement raisonnable, il occupera même pas 1 mégabits de mémoire vive sur un ordinateur. C'est pour cette raison qu'il n'est pas totalement absurde de vouloir représenter le jeu d'OXO sous forme d'arbre de jeu. Pour le jeu d'échecs, le nombre de noeuds avoisine les 10^{120}, ce qui est plus grand que le nombre de particules élémentaires présentes dans notre univers. Il va de soi que ce nombre est trop important pour pouvoir représenter un tel jeu sous forme d'arbre de jeu complet. Ceci explique pourquoi l'algorithme MinMax ne peut pas s'adapter aux jeux complexes.

3.3 Principe du MinMax

Ce second algorithme est un peu plus abstrait que le précédant, tout en restant logique et intuitif. L'idée est que, en tant que ordinateur, on va supposer que notre adversaire, que ce soit un ordinateur ou un humain, joue de façon optimale. Ainsi, on peut prédire son coup, et donc toute la partie par après, et jouer l'action qui nous arrange le plus.

Modélisons dès lors ceci. Commençons par dresser l'arbre de jeu et de considérer les feuilles de cet arbre. On va leur attribuer une valeur $v(x)$ suivant le joueur qui gagne dans la situation présente. Si le joueur "X" gagne, on va attribuer une valeur $+1$, si le joueur "O" gagne on va attribuer une valeur -1 et si personne ne gagne on va attribuer 0 simplement.

Le but du premier joueur, le joueur "X", est d'arriver à une situation finale valant $+1$ de préférence, et 0 dans le pire des cas. Le second joueur, jouant "O", veut arriver à une situation finale valant -1, ou si cela n'est pas possible, à une situation finale de valeur 0. Si, pour un des joueurs, il est dans un état tel qu'il existe une action lui permettant d'atteindre son but en un seul coup, il va le prendre forcément.

Pour assigner toutes ces valeurs, nous allons itérer sur chaque parent en partant des noeuds de profondeur maximale. La manière dont un parent va se voir attribuer sa valeur dépend de la parité de sa profondeur;

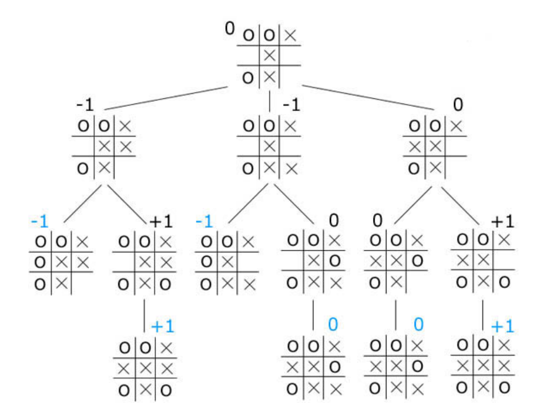
Si sa profondeur est paire, donc si la prochaine action sera prise par le joueur "X", alors la valeur du noeud sera le maximum de ses enfants. On peut donc écrire

$$v(x_k) = \max_{e \in e(x_k)} v(e)$$

Où x_k est un noeud de profondeur paire.

De façon analogue, si la profondeur d'un noeud est impaire, donc l'action suivante sera prise par "O", alors sa valeur sera le minimum des valeurs de ses enfants. Si on dénote par x_k le noeud de profondeur impaire, alors on a

$$v(x_k) = \min_{e \in e(x_k)} v(e)$$



3.4 Preuve du MinMax

Montrons que le joueur "X" a une stratégie non-perdante. Sa stratégie va être de toujours choisir l'action qui a le meilleur rendement pour lui dans l'arbre de jeu suivant les valeurs qu'on a assigné à chaque état plus haut.

Pour montrer que cette stratégie peut en effet empêcher au joueur "X" de perdre, il nous faut introduire deux propriétés très similaires.

Propriété 1: Si la valeur d'un état de jeu x_i vaut $+1$ à un moment donné, alors le joueur "X" peut gagner s'il joue bien.
Propriété 2: Si la valeur d'un état de jeu x_i vaut -1 à un moment donné, alors le joueur "O" peut gagner s'il joue bien.

Vu la similarité des propriétés, on ne va en démontrer qu'une, mais la démonstration de l'autre est totalement analogue.

Preuve Propriété 1: Soit x_{i+1} tel que $x_i \to x_{i+1}$. Nous allons partager la preuve en deux parties: Les actions commises par "O" et celles par "X".
-Si c'est au tour de "O", alors nous savons que

$$+1 = v(x_i) = \min_{e \in e(x_i)} v(e)$$

Or, comme $v(x_j) \in \{-1, 0, +1\}$ pour tout j, on sait que, pour tout ensemble A,

$$\min_{a \in A} v(x_a) = +1 \iff v(x_a) = 1 \quad \forall a \in A$$

Dès lors on en déduit que

$$e \in e(x_i) \implies v(e) = +1$$

Et ainsi, peu importe l'action choisie par "O", l'état du coup x_{i+1} aura également une valeur de +1.
-Si au contraire c'est au tour de "X", alors nous savons que

$$+1 = v(x_i) = \max_{e \in e(x_i)} v(e)$$

Cela implique que

$$\exists e \in e(x_i) \text{ tel que } v(e) = +1$$

Ainsi il suffit au joueur "X" de choisir cet enfant et ainsi de garder la valeur de +1 à l'état de jeu.

De ce fait le joueur "X" peut jouer de telle façon que la valeur de l'état de jeu ne changera plus et vaudra toujours +1. Comme dans notre cas le jeu est fini, cela implique que le jeu va se terminer dans un état de valeur +1, ce qui implique, par définition, que le joueur "X" a gagné. □

De ce fait, on sait que dès que un des états de jeu a une valeur différente de 0, le jeu est perdu pour un des joueurs. Nous allons montrer une dernière propriété qui va nous aider à conclure que le joueur "X" possède une stratégie non-perdante.

Propriété 3: Supposons que la valeur d'un état de jeu x_i vaut 0 à un instant donné. Le mieux que le joueur puisse faire est une action gardant la valeur du jeu à 0.

Preuve Propriété 3: Nous allons de nouveau définir x_{i+1} comme l'état suivant x_i, donc $x_i \rightarrow x_{i+1}$. Nous allons traiter deux cas suivant le joueur:
-Si c'est au tour du joueur "X", alors on a

$$0 = x_i = \max_{e \in e(x_i)} v(e)$$

Ainsi, on sait que

$$\nexists e \in e(x_i) \text{ tel que } v(e) = +1$$

Et que

$$\exists e \in e(x_i) \text{ tel que } v(e) = 0$$

Il se peut également qu'il existe e un enfant de x_i tel que sa valeur vaille -1, bien que cela ne soit pas obligé. Par la propriété 2, si le joueur "X" décide de jouer un état dont la valeur vaut -1, il a perdu le jeu. Ainsi, le mieux qu'il puisse faire est maintenir la valeur du jeu à 0.
–Si c'est au tour du joueur "O", la preuve est analogue. □.

Pour terminer cette preuve que l'algorithme MinMax apporte bel et bien une stratégie optimale, c'est à dire non-perdante, pour le joueur "X", il nous faut déterminer la valeur de l'état d'une grille vide. Pour ce faire, nous aurons recours à un ordinateur qui parcourt l'entièreté de l'arbre pour ce déterminer. On apprend très rapidement que la valeur initiale est 0, et qu'il existe donc une stratégie non-perdante pour le joueur "X", notamment celle expliquée plus haut. Une propriété intrigante du jeu d'OXO est qu'il existe également une stratégie non-perdante pour le joueur "O", et que la stratégie est la même que pour le joueur "O". Ainsi, l'algorithme MinMax apporte aux deux joueurs une stratégie non-perdante, et à aucun joueur une stratégie de victoire stricte.

3.5 Généralisation

La preuve du point 4.4 n'a rien de spécifique au jeu d'OXO, nous l'avons uniquement pris comme exemple pour simplifier la compréhension. En effet, le joueur "X" peut facilement être remplacé par un joueur "1", et le joueur "O" par un joueur "2". L'idée reste la même, c'est à dire qu'on remplit l'arbre de jeu suivant l'idée que le joueur 1 veut maximiser sa valeur et le joueur 2 veut la minimiser, chacun pour se rapprocher le plus possible de leurs buts finaux. Il y avait tout de même quelques idées importantes dans la preuve qui fait que ceci ne s'adapte pas à tout jeu.
Pour commencer il faut que le jeu soit à information complète, ce qui signifie que les deux joueurs savent tout ce qui se passe à tout moment. Ainsi, les deux joueurs peuvent dresser l'arbre de jeu et suivre la stratégie décrite plus tôt.
Il faut également que le jeu soit déterministe, ce qui veut dire que si les deux joueurs adoptent la même stratégie plusieurs jeux de suite, le rendement sera le même. Ceci est équivalent à ce que aucun facteur de chance soit impliqué dans le jeu.
Sous ces contraintes, nous savons que au moins un des joueurs a une stratégie non-perdante.

Le résultat que nous venons de montrer porte un nom; Théorème de Zermelo: Dans tout jeu fini à deux joueurs, à information parfaite et sans hasard, un des deux joueurs a une stratégie non-perdante.

Ainsi, nous n'avons pas seulement montré que sous ces contraintes il existait une stratégie non-perdante pour un des joueurs, mais nous avons même explicité une telle stratégie. Ceci nous apprend par exemple qu'il existe un joueur ayant une stratégie gagnante aux échecs, et qu'il nous est même possible de déterminer quel joueur c'est et de lui décrire une stratégie en théorie.
Pourtant, comme décrit au point 4.1, il nous est impossible d'évaluer l'arbre de jeu des échecs, donc il n'est pas possible de déterminer la valeur du noeud initial. Ainsi, on n'a aucune chance de pouvoir décrire une stratégie optimale, et on ne peut même pas déterminer lequel des joueurs a une stratégie non-perdante. Le résultat est extrêmement intéressant en théorie, mais impossible à appliquer en pratique, ce qui est bien dommage.

4 Apprentissage par renforcement

4.1 Brève introduction

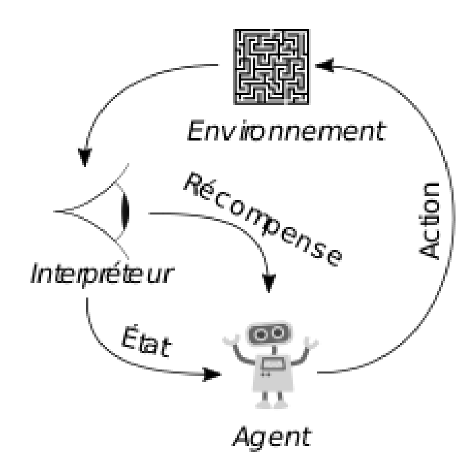
L'apprentissage par renforcement est l'un des domaines les plus prometteurs de l'apprentissage automatique. Ceci est dû à l'étendue des applications de ces algorithmes et les récents succès apportés par ceux-ci. Prenons l'exemple d'AlphaGo, un logiciel conçu par DeepMind technologies, qui a surpassé l'être humain au jeu de Go. Ce dernier est considéré comme un des jeux les plus complexes d'aujourd'hui et la victoire de la machine souligne la magnitude des dernières avancées technologiques.

La robotique est également un milieu où ce type d'apprentissage est utilisé. De nos jours, les robots sont physiquement parfaitement capables d'accomplir un large éventail de tâches utiles. Cependant ils nous est souvent difficile voir impossible leur fournir "l'intelligence" nécessaire pour effectuer ces taches. La création d'une robotique de pointe est un défi logiciel et non un problème matériel. Il est ironique qu'un robot puisse effectuer des millions de calculs par seconde très facilement, ce qui est totalement impossible pour un être humain, mais éprouve des difficultés à apprendre à faire quelque chose de très simple pour un humain, comme ramasser une bouteille. L'apprentissage par renforcement offre des belles perspectives d'avenir.

4.2 Mots de vocabulaire

Commençons par introduire quelques mots de vocabulaire qui pourront s'avérer utile par la suite:
• Un *Agent* est le joueur, en général il est juste un objet quelconque capable de prendre des décisions.
• L'*Environnement* est le milieu avec lequel l'agent interagit.
• Une *Action* est prise par un agent dans un environnement donné, et qui va le modifier en fonction du type d'action choisie.
• L'*État* est la position dans laquelle se trouve l'environnement après un certain nombre d'actions.
• La *Stratégie* détermine les actions pour un état donné. Elle peut être fixe ou adaptée au fur à mesure que l'algorithme apprend par lui-même.
• La *Récompense* est ce qui reçoit l'agent après avoir réalisé une action. Celle-ci peut être positive si l'action engendre un bon état ou négative si l'action engendre un état défavorable.

Dans une situation d'apprentissage par renforcement (AR), un agent se trouve dans un état initial et a le choix entre plusieurs actions qui modifient son environnement. Dépendant de sa stratégie, il effectue certaines actions qui entrainent une récompense. Celle-ci peut être positive ou négative en fonction que l'état obtenu soit considéré comme plus ou moins avantageux pour l'agent. Le but de l'agent est de maximiser le nombre de récompenses positives découvrant ainsi la stratégie optimale[2]. C'est dans ce cadre que l'on peut modéliser la plupart des jeux.



Source : Wikipédia

En pratique, la modélisation mathématique de l'agent, de l'environnement et des actions relève plus de la routine que de l'innovation. En revanche, dans des jeux plus complexes tels les échecs, il devient compliqué d'évaluer si un état est en faveur d'un agent ou d'un autre et d'en accorder les récompenses appropriées.

4.3 L'Équation de Bellman

Considérons un agent dans un état donné et supposons qu'il prendra toutes les actions optimales possibles à partir de cet état. L'équation de Bellman permet de trouver quelle sera la récompense à long terme à partir de cet état. En résumé, est-ce qu'en appliquant la meilleure stratégie possible cet état est bon ou mauvais ? Pour un environnement déterminé (pas de composante aléatoire) l'équation de Bellman est :

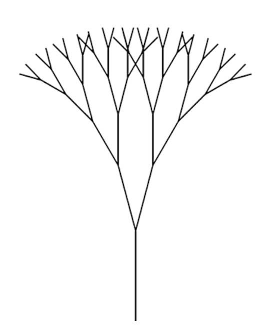
$$V(s) = \max_a (R(s,a) + \gamma V(s'))$$

où $V(s)$ représente la valeur de l'état actuel s, $R(s,a)$ la récompense associée à une action a dans l'état s, γ est le facteur de réduction des récompenses[3], et $V(s')$ est la valeur de l'état suivant s'. Le $\max_a$ représente le choix de l'action qui maximise la valeur de $(R(s,a) + \gamma V(s'))$. Cette équation fournit une expression récursive de $V(s)$. Pour utiliser de manière simple cette équation, on commence par calculer toutes les valeurs des états finaux possibles avant de remonter l'arbre de récursion et d'obtenir la valeur pour l'état actuel. Cette technique est utile dans des situations où les états finaux ne sont pas nombreux mais devient

[2]Les algorithmes très performants valorisent une stratégie adaptative dépendant de la situation rencontrée. Dans les deux prochaines sous-sections nous allons établir les bases et considérer que l'agent a parfaite connaissance de la stratégie optimale. Ensuite nous allons nous intéresser de plus prêt à la stratégie pour transiter vers les méthodes de Monte-Carlo qui privilégient une découverte de la stratégie par l'agent.

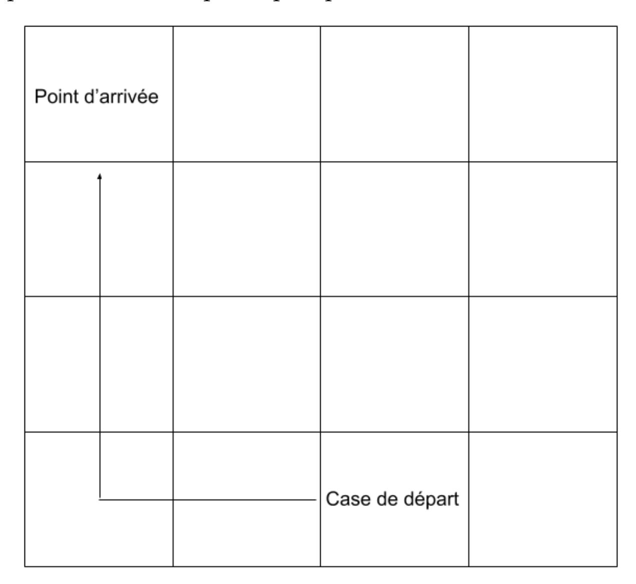
[3]C'est un super-paramètre qui permet de réduire les récompenses au fur à mesure qu'on s'éloigne.

inutilisable dès que le nombre de possibilités devient trop grand.



Cet algorithme est un des premiers à utiliser le principe de récursion et son inventeur, Richard Bellman, est considéré comme le fondateur de la programmation dynamique.

Considérons une application concrète de l'équation de Bellman dans le problème suivant. On est en présence d'une grille 4×4 et on a un robot qui se trouve dans une des 16 cases et on veut qu'il se déplace dans la case en haut en utilisant les lignes et les colonnes. Quel procéder utiliser pour qu'il puisse trouver son chemin?



On peut utiliser l'équation de Bellman pour associer à chaque case (chaque case représente un état) une certaine valeur. On commence à partir d'une case adjacente à l'arrivée, $(2,1)$, et on calcule de manière récursive pour les autres cases. Par exemple pour la case $(2,1)$ on trouve que :

$$V((2,1)) = 1 + \gamma \cdot 0$$

en prenant pour convention que la récompense d'une action amenant le robot sur l'arrivée est définie comme 1 et la valeur de la case d'arrivée vaut 0 qui est une valeur terminale. Pour γ on prend une valeur plus petite que 1 afin que celle-ci diminue pour chaque case au fur à mesure qu'on s'éloigne du point d'arrivée. Typiquement on prend une valeur $\gamma \in [0.9, 0.99]$. Posons $\gamma = 0.9$, et calculons $V((3,1))$:

$$V((3,1)) = 0 + 0.9 \cdot 1$$

car il n'y a pas de récompense pour atteindre la case $(2,1)$ et la valeur de $(2,1)$ vaut 1 (on l'a calculé avant). La grille d'en-dessous permet de visualiser les valeurs de chaque cases obtenues une fois la récursion finie. Nous avons arrondi les valeurs à deux chiffres significatifs.

Point d'arrivée	1	0.9	0.81
1	0.9	0.81	0.73
0.9	0.81	0.73	0.66
0.81	0.73	0.66	0.59

Le robot n'as donc plus qu'à choisir ses actions pour se déplacer vers une case ayant une valeur plus élevée et finira par arriver à sa destination.

Il est important de noter que l'équation de Bellman permet uniquement d'établir un cadre mathématique pour l'apprentissage par renforcement. C'est dans le choix de l'action a qui doit maximiser le terme de droite de l'équation que réside toute la subtilité. Dans ce cas simple, plus les cases sont proches de l'arrivée, plus elles sont considérées comme avantageuse pour l'agent.

4.4 Succession finies de décisions de Markov

Dans la section précédente, nous avions supposé que l'agent prendra toutes les actions optimales à partir d'un certain état. Cela signifie que l'agent ne peut pas se tromper et cela est vrai dans un environnement idéalisé. Cependant dans un mondé plus réaliste il existe toujours une probabilité que l'agent ne choisisse pas le coup optimal en toute "conscience". Comment exprimer cette stochasticité en termes d'équations ? Les successions de décisions de Markov (SDM) fournissent une réponse. Pour qu'un jeu puisse être considéré comme une SDM, chaque état du jeu doit vérifier la propriété de Markov qui dit :

Définition : En probabilité, un processus stochastique vérifie la propriété de Markov si et seulement si la distribution conditionnelle de probabilité des états futurs, étant donnés les états passés et l'état présent, ne dépend que de l'état présent et non pas des états antérieurs.

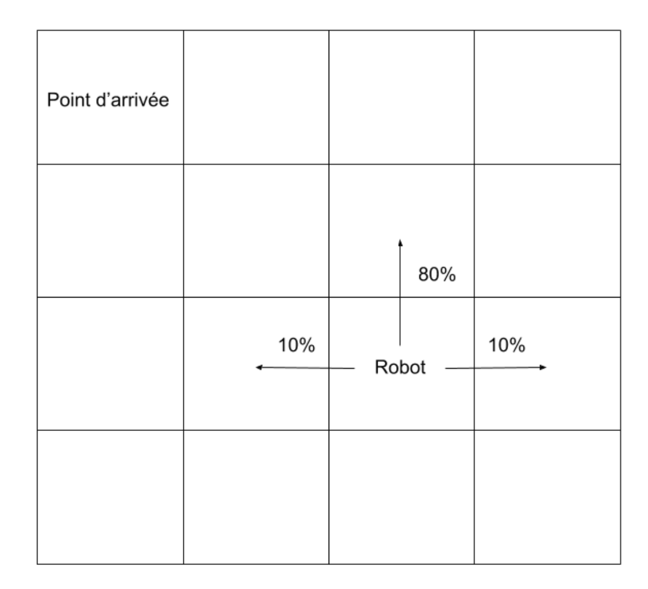
En d'autres termes, pour prendre une action "réfléchie" dans un état donné il n'est pas nécessaire de prendre en compte tous les états précédents (il n'y a pas de mémoire). On va s'intéresser aux jeux finis ce qui veut dire que le nombre d'états, d'actions et de récompenses possibles est fini. Si un jeu fini respecte la propriété de Markov alors il peut être considéré comme une succession finie de décisions de Markov (SFDM). Des jeux connus qui respectent ces conditions sont "Snakes and Ladders", "Hi Ho! Cherry O!" et même les règles du célèbre jeu "Monopoly" ont été adaptées au modèle de Markov.

Dans l'équation de Bellman nous avons le terme $V(s')$ qui représente la valeur de l'état s' après une action amenant l'agent de l'état s à l'état s'. Cependant dans une SFDM, une même action peut nous amener à plusieurs états différents s' avec une probabilité respective. Nous pouvons remplacer le terme $V(s')$ par la somme des probabilités qui gouvernent la transition pour une action a d'un état s à un état s' multiplié par la valeur de chaque état s', ce qui donne la version stochastique de l'équation de Bellman :

$$V(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} P\left(s,a,s'\right) V\left(s'\right) \right)$$

où $P(s,a,s')$ est la probabilité que l'action a amène l'agent de l'état s à l'état s'.

Grâce à cette équation nous pouvons résoudre des problèmes qui ont une composante aléatoire. Reprenons l'exemple du robot dans la grille 4×4. Pour intégrer la notion de stochasticité on va supposer que lorsque le robot doit monter ou descendre il le fera dans 80% des cas et dans 10% des cas il ira à gauche ou à droite. De la même manière lorsqu'il doit aller à gauche ou a droite il le fera dans 80% des cas et dans 10% des cas il ira vers le haut ou vers le bas.



Il n'est désormais plus possible de calculer la valeur de chaque case à la main. Il faut donc faire appel à un ordinateur pour effectuer la récursion et les calculs. L'algorithme utilisé est celui-ci qui appartient à la classe des algorithmes de programmation dynamique :

1. On crée un tableau V avec les estimations des valeurs de chaque case et associe à chaque case la valeur 0.
2. **for** *chaque état s possible* **do**
   3. **for** *chaque action a possible* **do**
      4. On calcule les probabilités associés à la transition de l'état s à l'état s' résultant de l'action a ce qui équivaut à calculer $P(s,a,s')$.
      5. La récompense $R(s,a)$ vaut la somme des récompenses possibles multipliées par leur probabilité respective.
      6. La valeur attendue de l'état vaut la somme des $V(s')$ multiplié par la probabilité respective.
      7. La valeur de l'action a vaut la somme de $R(s,a)$ et de valeur attendue de l'état multiplié par le facteur de réduction γ (équation de Bellman).

   end
   8. On associe à $V[s]$ la plus grande valeur parmi toutes les action possibles.

end

9. On répète les étapes 2-8 jusqu'à ce que le plus grand changement du tableau V soit inférieur à un seuil prédéfini.

En faisant tourner cet algorithme sur un ordinateur, en prenant $\gamma = 0.9$ et un seuil de 10^{-2} on obtient le résultat suivant pour la valeur de chaque état :

Point d'arrivée	0.93	0.67	0.45
0.93	0.69	0.49	0.31
0.67	0.49	0.31	0.16
0.45	0.31	0.16	0.04

On peut faire plusieurs observations :

- La symétrie de la configuration est bien respectée.
- Les cases juste à côté du point d'arrivée n'ont plus la valeur 1 vu qu'il existe une probabilité que le robot prenne la mauvaise direction.

- Les valeurs des cases décroissent beaucoup plus vite. Si on utilise le même algorithme sur une grille plus grande il serait judicieux d'augmenter légèrement la valeur de γ pour palier à ce problème.

4.5 La stratégie

Nous avons fini les deux sous-sections précédentes en donnant le tableau des valeurs de chaque état et nous nous sommes contentés de dire que le robot n'a qu'à se diriger vers les états ayant des valeurs plus élevées pour arriver à destination. Pour arriver à exprimer ce comportement de manière formelle il faut s'intéresser à la stratégie.

Définition : De manière générale, une stratégie π est une association entre les états s et la probabilité de prendre une action a.

Par exemple, si un agent suit la stratégie π alors $\pi(a|s)$ représente la probabilité que l'agent choisisse l'action a s'il se trouve dans l'état s. Par ailleurs on définit $Q(s, a)$ comme la récompense à long terme espérée pour avoir choisi l'action a à l'état s. Par définition on a que :

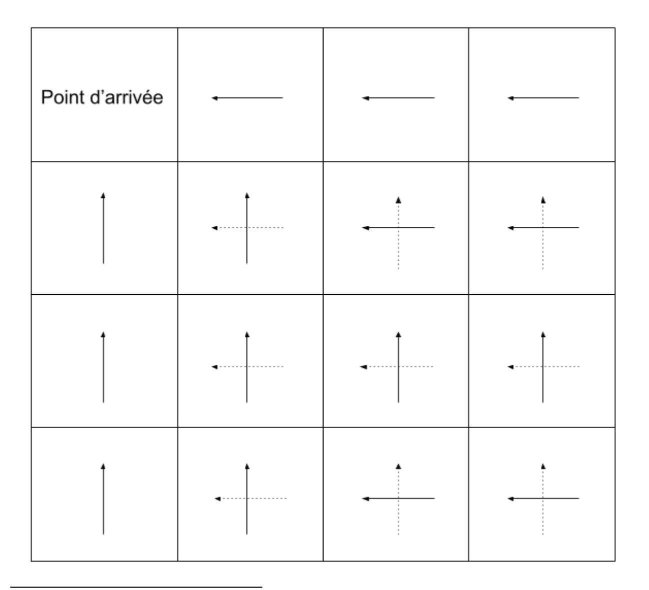
$$Q(s, a) = \mathbb{E}[R(s, a) + \gamma V(s')]$$

Par conséquent, la stratégie optimale à l'état s vaut :

$$\pi(s) = \text{argmax}_a(Q(s, a))$$

la stratégie optimale π ayant pour but de maximiser le gain total à partir de l'état s[4].

Dans l'algorithme présenté à la sous-section précédente, nous avons calculé de manière directe toutes les valeurs des états possibles (représentés dans le tableau de la page 11). Nous pouvons dès lors calculer la stratégie optimale à partir de la formule :



Dans cette grille les actions optimales enregistrées par la stratégie sont représentée graphiquement sous forme de flèche. Bien entendu dans le programme informatique elle correspondent à des incrémentations des indices de la case dans laquelle se trouve le robot. On observe également que pour le bloc 3 situé en bas à droite il y a deux possibilités d'actions optimales. La stratégie que nous avons implémenté ne compte qu'une action optimale pour chaque état et les flèches pleines la représentent. Les flèches en pointillés correspondent à des variantes de la stratégie optimale que l'algorithme aurait pu calculer.

Jusqu'à présent, les algorithmes qu'on a développé, utilisent la connaissance parfaite de chacun des états du jeu pour trouver la stratégie optimale avant même d'y avoir joué. Ceci est une approche dite avec un modèle fixe. On peut la différencier des approches avec un modèle adaptatif, où l'on prend une stratégie de base et l'agent l'améliore au fur à mesure qu'il gagne de l'expérience avec le jeu. La notion d'expérience est clé dans la réalisation d'algorithmes performants et la sous-section suivante présente une nouvelle approche.

4.6 Apprentissage Monte Carlo

Le problème principal de la programmation dynamique est le besoin inné d'avoir toute l'information nécessaire pour prendre des décisions. L'apprentissage Monte Carlo se distingue de cette dernière en incorporant une composante aléatoire dans la prise de décision[5]. La différence est qu'on ne travaille plus avec des valeurs exactes mais des estimations. En partant d'une stratégie aléatoire nous allons arriver à certaines prédictions concernant les valeurs de chaque état. Ensuite avec ces nouvelles valeurs on va pouvoir adapter la stratégie. En effectuant ce processus d'évaluation de la stratégie actuelle et d'adaptation en fonction des nouvelles valeurs des états un grand nombre de fois, la valeur des états va converger vers leur vraie valeur de même que la stratégie va converger vers la stratégie optimale. Ce principe s'appelle "General Policy Iteration" et s'applique pour la plupart des algorithmes d'apprentissage par renforcement. Pour ne pas casser le fil conducteur nous fournirons les bases théoriques en fin de section.

Prenons l'exemple de notre robot dans la grille 4×4 avec un algorithme sans modèle où l'agent part avec une stratégie aléatoire. Le but de l'agent est de trouver par essai-erreur la meilleure stratégie possible qui dans ce cas-ci sera même optimale. Pour ce faire nous allons modifier le fonctionnement des récompenses et ajouter la notion de décisions aléatoires.

Auparavant, nous avons offert une récompense quand

[4] La fonction argmax renvoie simplement la valeur de l'argument qui maximise la fonction.

[5] Il ne faut pas confondre la notion de stochasticité de l'environnement avec une composante aléatoire dans la prise de décision. En effet cette dernière n'est pas le résultat d'un dysfonctionnement du système mais bien le résultat d'une décision volontaire de choisir une action aléatoire.

le robot arrive sur la case d'arrivée qui correspond à l'état terminal. Maintenant nous allons offrir les récompenses au fur à mesure de l'avancement dans le jeu. Pour ce faire, introduisons la notion de retour :

Définition : Le retour G est la somme de la récompense pour l'action actuelle dans l'état s ainsi que de toutes les récompenses futures multipliées par le facteur de réduction des récompenses γ en suivant la stratégie actuelle π.

Ce qui s'exprime en mathématiques :

$$G_t = r_t + \gamma \cdot G_{t+1}$$

où r_t est la récompense à l'étape[6] t. Il en ressort que $Q(s,a) = \mathbb{E}[G_t \mid S_t = s, A_t = a]$.

Sous forme d'algorithme, on peut trouver la valeur des paires (état, retour) à partir de la valeur des paires (état, récompense) selon l'algorithme suivant :

1. On initialise la valeur de la variable G à 0.
2. On crée un tableau état_retour.
3. **for** *chaque paire (état, récompense), en partant de la fin* **do**
   - 4. On ajoute au tableau à état_retour la valeur de (s, r).
   - 5. $G = r + \gamma \cdot G$ (on remonte en arrière)

   end
6. On inverse l'ordre du tableau état_retour.

Le principe des techniques Monte-Carlo est de trouver la valeur de chaque état s. Pour ce faire nous calculons la valeur attendue du retour G pour chaque paire d'action (s, a) qui est en fait $Q(s, a)$, puis nous assignons la valeur la plus élevée parmi les différents retours à $V(s)$.

En ce qui concerne les décisions aléatoires, on doit faire face à un dilemme. Étant donné une stratégie π, comment faire en sorte que l'agent explore suffisamment son milieu (en faisant des coups aléatoires) tout en utilisant raisonnablement l'expérience acquise pour améliorer sa stratégie.

Une solution à ce problème est d'utiliser une technique appelée *epsilon-greedy*. L'idée est simple : dans $1 - \epsilon$ des cas on choisi l'action optimale selon la stratégie suivie et dans ϵ des cas on prend une décision aléatoire avec $\epsilon \in (0, 1)$.

Le choix du super-paramètre ϵ dépend de l'orientation que l'on veut donner à l'algorithme. Un ϵ plus grand entraine une plus grande exploration, tandis qu'un ϵ plus petit privilégie la connaissance acquise dans la stratégie.

Pour finir, il est prouvé qu'il n'est pas nécessaire de prendre en compte plusieurs fois les paires état/action de le même épisode. Cet algorithme s'appelle un Monte-Carlo "première visite" qui se distingue du Monte-Carlo "chaque visite" par le nombre de visite d'une paire état/action pendant le même épisode. Cette "astuce" permet d'économiser un peu de puissance de calcul.

En combinant tout ce qu'on a appris dans cette sous-section, on peut créer un nouvel algorithme :

1. On initialise une stratégie π avec une action aléatoire pour chaque état.
2. On initialise le tableau $Q[(s, a)]$ a 0 pour tout paire d'état s et d'action a.
3. On initialise retour comme un tableau vide pour toute paire (état, action).
4. **for** *une valeur N, assez grande pour converger* **do**
   - 5. On joue le jeu jusqu'à la fin en suivant la stratégie π dans $1 - \epsilon$ des cas et on reçoit la liste des état_action_retour.
   - 6. **for** *pour chaque tuplet (S, A, G)* **do**
     - **if** *Si la paire (S, A) n'a pas encore été visitée* **then**
       - 7. On ajoute dans le tableau retour la valeur de G pour l'action A dans l'état S.
       - 8. $Q[S][A] = \text{moyenne } (\text{retour}[(S, A)])$

       end

     end
   - 9. **for** *chaque état non terminal S* **do**
     - $\pi[S] = \text{argmax}_a(Q[S])$

     end

   end
10. **for** *chaque état S* **do**
    - $V[S] = \max(Q[S])$

    end
11. On revoie la tableau V et la stratégie π finale.

Quelques précisions s'imposent :

Premièrement, la valeur N qui représente le nombre de parties jouées nécessaires pour arriver à une approximation assez bonne, dépend de la vitesse de convergence de l'algorithme et de la taille de la liste état_action_retour (la taille du jeu). Nous discuterons de la vitesse de convergence plus loin, mais il est utile de savoir que celle ci dépend fortement de la valeur d'epsilon qui gouverne le rapport exploration/exploitation.

Deuxièmement, le lecteur attentif aura remarqué que

[6] On différencie les tâches épisodiques des taches continues. La majorité des jeux sont épisodiques, c'est à dire qu'ils ont un début et une fin qui délimitent la succession d'étapes intermédiaires. Nous définissons un épisode comme faire une partie du jeu considéré. Il est courant de jongler entre les notations à une étape (avec un indice t) et celle générales qui ne concernent pas une étape (sans l'indice)

cet algorithme fournit une stratégie déterminée et non stochastique ce qui signifie que lorsque l'agent a suivi sa stratégie il a choisi l'action qui maximise le retour à long terme à raison de 100%. Si nous voulons inclure la notion de "dysfonctionnement" de l'agent de la sous-section 4.4, nous pouvons procéder de deux manières :

- Soit nous pouvons supposer que l'entraînement est dans environnement idéal et utiliser la stratégie déterminée trouvée par l'algorithme ci-dessus pour ensuite l'appliquer dans un environnement stochastique.
- Soit il faut modifier légèrement l'algorithme pour qu'il tienne compte d'un entrainement dans un environnement stochastique pour l'appliquer également dans un environnement au mêmes conditions.

Le deuxième choix est clairement plus avantageux car il permet de mieux simuler les conditions réelles rencontrées et d'obtenir une meilleure approximation de la valeur de chaque état. Cependant il existe des situations où la stochasticité est tellement petite qu'il est tout à fait acceptable d'utiliser une stratégie déterminée.

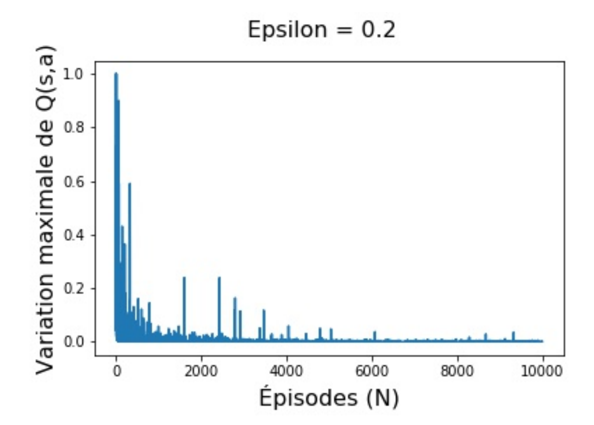
Avec ces remarques en tête, nous pouvons désormais analyser les résultats données par l'algorithme (la version déterministe). Voici le tableau des valeurs de chaque état pour $\epsilon = 0.2$ et $\gamma = 0.9$ et $N = 10^4$:

Point d'arrivée	1	0.87	0.72
1	0.87	0.76	0.63
0.87	0.77	0.67	0.58
0.76	0.67	0.59	0.53

En comparant ce tableau avec celui de la sous-section 5.2, nous observons quelques ressemblances :

- les valeurs sont plus au moins pareilles même si elles sont légèrement plus petites que celle de l'algorithme de programmation dynamique. Ceci est du au fait que l'algorithme joue des coups aléatoires dans ϵ% des cas ce qui fait baisser la valeur de l'état de chaque case. Étant donné que la valeur de toutes les cases diminue l'algorithme trouvera quand même la stratégie optimale.
- la symétrie de la configuration est globalement respectée.
- même si cette solution approximative n'est pas parfaite, elle est suffisamment précise pour donner de bons résultats.

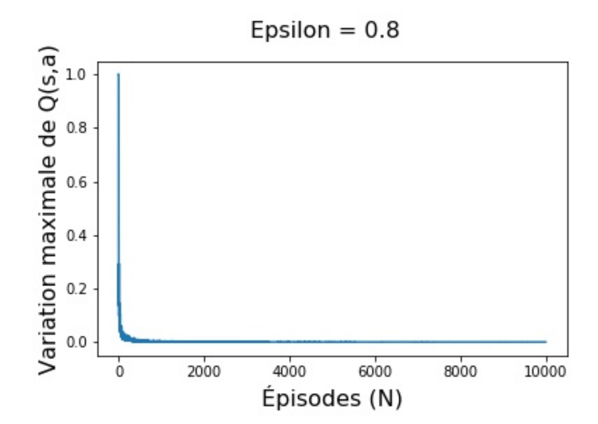
Le lecteur pourrait se poser la question de savoir combien de jeux sont nécessaire pour que le tableau des valeurs converge vers la solution optimale pour un ϵ donné. Pour ce faire il suffit de faire un graphique de la plus grande variation de $Q(a, s)$ en fonction du nombre d'épisodes.



Dans le graphique ci-dessus pour $\epsilon = 0.2$, nous observons que pour $N \geq 3000$ les fluctuations s'atténuent et on peut conclure que nous avons atteint une assez bonne approximation de la stratégie optimale.

Pour se rendre bien compte des changements opérés par une modification de la valeur de ϵ, considérons les deux figures ci-dessous avec $\epsilon = 0.8$ et $\gamma = 0.9$ et $N = 10^4$:

Point d'arrivée	1	0.64	0.42
1	0.64	0.46	0.34
0.63	0.46	0.34	0.26
0.42	0.34	0.26	0.21



En comparant ces graphiques à ceux pour $\epsilon = 0.2$, nous confirmons la tendance qu'ont les valeurs des états de baisser quand nous augmentons ϵ. De plus l'algorithme converge bien 5 ou 6 fois plus vite. Nous pouvons en conclure que dans ce cas spécifique il est plus avantageux de privilégier l'exploration.

De manière générale, il est intéressant de privilégier un algorithme avec un taux d'exploration adaptatif. Nous commençons avec une valeur d'epsilon élevée qui entraine une grande exploration, ensuite nous la diminuons au fur à mesure que les épisodes s'enchaînent. Une manière d'y arriver est d'exprimer ϵ comme :

$$\epsilon = \frac{\epsilon_0}{1 + \mathrm{e} \cdot t}$$

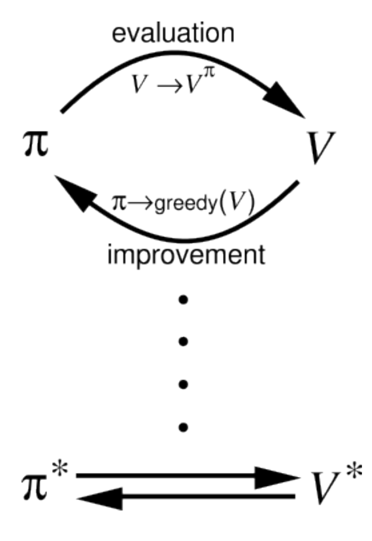
où e est une constante de réduction d'exploration proche de 0.

Pour finir nous pouvons calculer la vraie valeur des états en utilisation la technique d'évaluation de la stratégie ce qui donne le même résultat que celles trouvées par programmation dynamique dont le tableau est repris ici :

Point d'arrivée	1	0.9	0.81
1	0.9	0.81	0.73
0.9	0.81	0.73	0.66
0.81	0.73	0.66	0.59

4.7 General Policy Iteration

Le General Policy Iteration ou GPI est l'alternance de deux processus. Le premier consiste en la mise à jour des valeurs des états en fonction de la stratégie actuelle (évaluation de la stratégie). Le deuxième est de rendre la stratégie actuelle "greedy"[7] par rapport aux valeurs des états (amélioration de la stratégie). Ceci est résumé dans la figure suivante :



Source : [19]

Dans cette partie, nous démontrons pourquoi le GPI arrive à trouver la stratégie optimale et donc vraie valeur des états. Cette partie est fortement inspirée du chapitre 4.2 de [19].

Supposons avoir évalué la stratégie déterministe aléatoire π à la façon Monte-Carlo ou d'une autre manière. La question est de savoir s'il vaut mieux prendre pour un état s l'action de $\pi(s)$ ou de choisir une autre action $a \neq \pi(s)$ et de modifier la stratégie. Une manière de procéder est de choisir une action a parmi celles possibles à l'état s et d'ensuite suivre la stratégie π, on peut écrire cela comme :

$$Q_\pi(s, a) = \mathbb{E}[R(s, a) + \gamma \cdot V_\pi(s')]$$

La question est est-ce que la valeur $Q_\pi(s, a)$ est plus grande que celle de $V(s)$? Si c'est le cas, alors il serait logiquement plus avantageux de choisir l'action a au lieu de suivre la stratégie quand on se trouve à l'état s. Nous modifierions alors la stratégie à l'état s pour prendre la valeur de a. C'est justement ce que nous garantit le Policy Improvement Theorem[8] :

<u>Policy Improvement Theorem</u> : Soit π et π' n'importe quelle paire de stratégies déterministes (le résultat se généralise aussi aux stratégies stochastiques) telles que

$$Q_\pi(s, \pi'(s)) \geq V_\pi(s)$$

[7] Cela signifie que nous adapterons la stratégie de telle sorte à maximiser la valeur du retour pour un état.
[8] Il suffit de prendre π et π' deux stratégies identiques différant uniquement de l'action à l'état s et d'utiliser la version du théorème avec les inégalités strictes.

alors la stratégie π' est au moins aussi bonne que π, ce qui signifie que

$$V_{\pi'}(s) \geq V_\pi(s)$$

Le théorème est également vrai en remplaçant les deux inégalités par des inégalités strictes.

<u>Preuve</u> :

$$\begin{aligned}
V_\pi(s) &\leq Q_\pi(s, \pi'(s)) \\
&= \mathbb{E}[R(s, \pi'(s)) + \gamma \cdot V_\pi(s')] \\
&\leq \mathbb{E}_{\pi'}[R(s, \pi'(s)) + \gamma \cdot Q_\pi(s', a)] \\
&= \mathbb{E}_{\pi'}[R(s, \pi'(s)) + \gamma \cdot \mathbb{E}[(R(s, \pi'(s'))+ \\
&+ \gamma \cdot V_\pi(s''))]] \\
&= \mathbb{E}_{\pi'}[R(s, \pi'(s)) + \gamma \cdot R(s, \pi'(s')) + \gamma^2 \cdot V_\pi(s'')]
\end{aligned}$$

ce qui peut s'écrire avec la notation en étape t :

$$\begin{aligned}
&= \mathbb{E}_{\pi'}[R_t + \gamma \cdot R_{t+1} + \gamma^2 \cdot V_\pi(S_{t+2})] \\
&\leq \mathbb{E}_{\pi'}[R_t + \gamma \cdot R_{t+1} + \gamma^2 \cdot R_{t+2} + \gamma^3 \cdot V_\pi(S_{t+3})] \\
&\ldots \\
&= \mathbb{E}_{\pi'}[R_t + \gamma \cdot R_{t+1} + \gamma^2 \cdot R_{t+2} + \gamma^3 \cdot R_{t+3} + \ldots] \\
&= V_{\pi'}(s)
\end{aligned}$$

$\square$

Pour l'instant nous avons considéré l'impact du changement d'une action a à l'état s d'une stratégie π. Si on considère tous les états possibles avec toutes les actions possibles en choisissant chaque fois la meilleure action pour l'état s, nous obtenons la stratégie "greedy" suivante :

$$\pi'(s) = \text{argmax}_a(Q_\pi(s, a))$$

qui respecte par définition l'hypothèse du Policy Improvement Theorem. Nous en déduisons que la stratégie π' obtenue est au moins aussi bonne que la stratégie initiale π. Cela explique la partie amélioration de stratégie du GPI. Si nous avons une nouvelle stratégie greedy π' qui est aussi bonne que l'ancienne, cela implique que $V_\pi(s) = V_{\pi'}(s)$ et donc que

$$V_{\pi'}(s) = \max_a(Q(a, s)) = V(s)$$

par l'équation de Bellman où $V(s)$ représente la valeur des état selon la stratégie optimale. Nous en déduisons que π et π' sont toutes les deux des stratégies optimales ce qui implique l'amélioration de la stratégie ne s'arrêtera pas tant que l'optimalité n'est pas atteinte. Cela prouve le principe de convergence du GPI.



Source : xkcd

5 Q-Learning

5.1 Temporal Difference learning

Inventé en 1989, le Q-learning ou Quality-learning, est un algorithme d'apprentissage par renforcement dit "off-policy"[9] qui utilise un nouveau principe s'appelant Temporal Difference learning. Nous allons commencer par introduire les nouveaux concepts utilisés dans le Q-learning pour ensuite nous diriger vers l'expression mathématique de ceux-ci. Nous finirons par présenter notre implémentation du Q-learning pour résoudre un problème concret.

Le Temporal Difference learning ou TD learning est une combinaison des méthodes classiques d'apprentissage par renforcement comprenant la programmation dynamique et les méthodes Monte Carlo. Comme les algorithmes Monte-Carlo, le TD learning utilise l'expérience acquise en jouant au jeu pour peaufiner ses prédictions. La différence majeure entre les deux algorithmes, est que le TD learning met à jour la valeur de chaque état après t étapes au lieu d'attendre jusqu'à ce que l'épisode soit fini comme le fait la technique Monte Carlo. Ceci est donc avantageux pour des taches où les épisodes sont très longs. Le Q-learning est un algorithme de TD(0) ce qui signifie que la valeur de chaque état est mise à jour en fonction des récompenses apportées par l'état suivant. La convergence de cet algorithme "myope" peut sembler contre intuitive mais il est prouvé que TD(0) converge bien vers la solution optimale. Dans la plupart des situations le TD learning converge même plus vite que les techniques Monte-Carlo.

5.2 Taux d'apprentissage

Un nouveau concept essentiel dans l'apprentissage par renforcement est le taux d'apprentissage α qui est compris entre $(0, 1]$. Ce dernier permet de gouverner la vitesse à laquelle les tableau $V(s)$ et $Q(s, a)$ vont être modifiés. Dans les algorithmes montrés précédemment, nous avons utilisé indirectement un taux d'apprentissage variable lorsque nous calculions la moyenne des retours à l'étape 8 de l'algorithme par Monte Carlo. Les fluctuation des valeurs de $Q(s, a)$ étaient ainsi amorties par la moyenne.

$$Q(S_t, A_t) = Q(S_t, A_t) + \frac{1}{n(s)}[G_t - Q(S_t, A_t)]$$

où $n(s)$ représente le nombre de première visites à l'état s.

La différence avec le Q learning est qu'il se base uniquement sur la récompense de l'étape suivante et non sur le retour qui exprime la récompense à long terme. Nous remplaçons donc la moyenne par un taux d'apprentissage suffisamment petit pour éviter les importantes fluctuations dues à des mises à jour du tableau Q mais pas trop petit sinon l'algorithme mettra trop longtemps à converger.

En math, nous remplaçons la valeur réelle du retour G_t par la valeur estimée du retour qui est donnée par $R_{t+1} + \gamma \cdot \max_a Q(S_{t+1}, a)$ ce qui donne :

$$Q(S_t, A_t) = Q(S_t, A_t) + \\ + \alpha \cdot [R_{t+1} + \gamma \cdot \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

Comme pour ϵ, il est souvent intéressant d'avoir une valeur de α décroissante au fur à mesure de l'enchaînement des épisodes . Nous pouvons appliquer la même technique qu'à la sous-section 4.6 pour obtenir :

$$\alpha = \frac{\alpha_0}{1 + \mathrm{a} \cdot t}$$

où α est une constante de réduction d'apprentissage a très proche de 0.

Désormais nous avons tous les éléments nécessaires pour établir l'algorithme de base du Q learning :

1. On initialise une stratégie π avec une action aléatoire pour chaque état.
2. On initialise le tableau $Q[(s, a)]$ a 0 pour tout paire d'état s et d'action a non-terminale.
3. **for** *une valeur N, assez grande pour converger* **do**
    - 4. Initialisation de l'état de départ s.
    - **while** *l'état S n'est pas terminal* **do**
        - 5. On choisit une action A à partir de l'état S en suivant la stratégie π dans $1 - \epsilon$ des cas.
        - 6. On arrive dans l'état S' et un reçoit une récompense R.
        - 7. $Q(S, A) = Q(S, A) + \alpha \cdot [R + \gamma \cdot \max_a Q(S', a) - Q(S, A)]$
        - 8. L'état actuel devient S'.
    - **end**
    - 9. **for** *chaque état non terminal s* **do**
        - $\pi[s] = \text{argmax}_a(Q[s])$
    - **end**

end

10. **for** *chaque état S* **do**
    - $V[S] = \max(Q[S])$

end

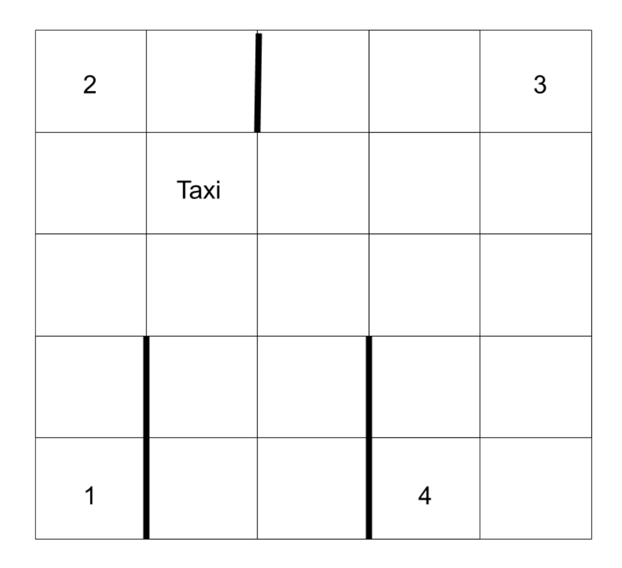
11. On revoie la tableau V et la stratégie π finale.

5.3 Exemple d'application

Considérons l'application suivante où nous sommes en présence d'une grille 5×5. Nous disposons d'un taxi et de 4 positions de départ ou d'arrivée. Le passager se situe dans l'une des 4 cases et doit arriver dans l'une des 3 cases restantes en respectant les contraintes les suivantes :

[9] Un algorithme off-policy à l'avantage de converger peut importe la stratégie qu'il suit. La seule condition pour la stratégie est que toutes les paires état/action soit visitées.

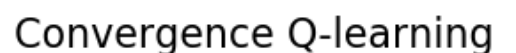
- Le taxi peut se déplacer selon les 4 points cardinaux
- Le taxi reçoit 20 points chaque fois qu’il accomplit un trajet correctement.
- Le taxi perd 1 point à chaque déplacement.
- Le taxi perd 10 point s’il prend ou dépose une personne à un mauvais endroit.

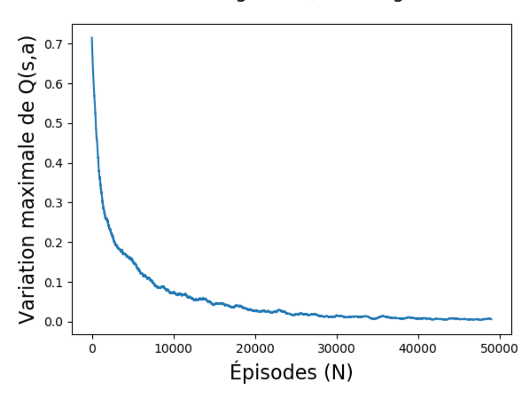


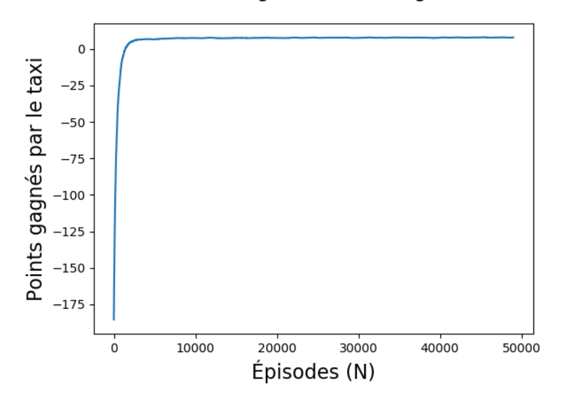
Quel est le trajet optimal pour que le taxi soit le plus efficace possible ?

Il y a 25 cases, 5 états possibles pour le passager (les 4 points arrivée/départ et dans le taxi) et 4 points d’arrivée/départ avec 6 actions possibles ce qui nous fait un tableau $Q(s, a)$ de taille 500×6. Vu la petite grandeur du tableau Q, nous pouvons facilement appliquer l’algorithme expliqué dans cette section.

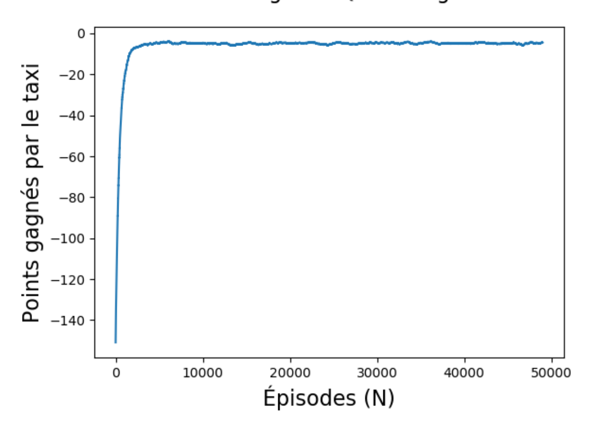
Il est impossible de montrer la valeur de chaque état car celle-ci n’est pas uniquement déterminée par la position de la case. Nous allons nous contenter d’observer le comportement général de l’algorithme en visualisant sa vitesse de convergence et la moyenne des points trouvée. Voici le graphique pour $\gamma = 0.95$, $\epsilon_0 = 1$, $e = 0.01$, $\alpha_0 = 0.1$, $a = 0.01$ et pour $N = 5 \cdot 10^5$:







En observant les figures, nous pouvons contempler la convergence de l’algorithme. Par ailleurs, la moyenne des points obtenus par le taxi est de 7.72 qui est proche de l’optimum. Il est important de noter que le choix des super-paramètres est déterminant pour obtenir des résultats optimaux. En variant légèrement les ces derniers nous pouvons obtenir des vitesses de convergence différentes. Un cas intéressant est de visualiser l’effet de l’absence du changement du taux d’exploration à travers le graphique suivant pour $\gamma = 0.95$, $\epsilon_0 = 0.2$, $e = 0$, $\alpha_0 = 0.1$, $a = 0.01$ et pour $N = 5 \cdot 10^5$:



Bien que l’algorithme converge vers une solution, celle-ci n’est absolument pas optimale vu que la moyenne des points obtenus par le taxi tourne aux alentours de -4.9. En général, nous modifions les super-paramètres pour obtenir la convergence la plus rapide tout en atteignant la solution optimale.

5.4 Développement futurs

Comme nous l’avons expliqué, Q-learning repose sur des calculs prenant en compte l’entièreté du tableau Q. Si celui-ci devient trop grand comme c’est le cas dans des jeux avec une multitude voire une infinité d’états/actions possibles[10], le Q-learning devient totalement inefficace et inutilisable. Ainsi, bien qu’il soit utile pour certaines applications, il n’est généralisable à merci. Par conséquent il a fallu trouver un nouveau modèle d’apprentissage automatique.

[10] Citons, entre autres, le jeu de dames, les échecs et le Go

6 Deep Learning

Le Deep Learning est, au contraire des autres méthodes exploitées plus tôt, un algorithme "intelligent". En effet, il ne consiste pas à tester tous les cas, que ce soit bêtement ou intelligemment, mais son idée s'inspire plutôt de l'intelligence humain. Ainsi, ce nouveau système apprend en inspirant d'exemples fournis auparavant.

6.0 Pré-requis

Pour comprendre cette section il faut quelques pré-requis en algèbre linéaire.

Tout d'abord, on appelle dérivée le taux de change d'une expression par rapport à une autre. On note la dérivée de f par rapport au paramètre x comme suit:

$$\lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x} = \frac{\delta f}{\delta x} = f'(x)$$

La deuxième notation est plus courante pour les dérivées plus simples, c'est à dire si la fonction n'admet qu'une unique variable. Lorsque le fonction admet plusieurs paramètres, il est important de préciser par rapport à quelle variable nous dérivons, et on parlera dès lors de dérivée partielle:

$$\lim_{\Delta x \to 0} \frac{f(x+\Delta x, y) - f(x,y)}{\Delta x} = \frac{\partial f(x,y)}{\partial x}$$

Lorsque la fonction prend plusieurs variables, nous pouvons définir son gradient, c'est à dire sa dérivée par rapport à un vecteur. Soit le vecteur $a = (a_1, a_2, \ldots, a_k)$, alors le gradient de f par rapport au vecteur a est égal au vecteur

$$\nabla_a f = \nabla_a f(a_1, a_2, \ldots, a_k) = \left(\frac{\partial f}{\partial a_1}, \frac{\partial f}{\partial a_2}, \cdots, \frac{\partial f}{\partial a_k}\right)$$

Par après, une matrice est un ensemble de nombres rangés dans un tableau rectangulaire. On peut par exemple noter la matrice $\mathcal{A}$ comme suit:

$$\mathcal{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

Elle a une dimension $n \times m$ (Nombre de lignes par nombre de colonnes).
Prenons une autre matrice de même dimension:

$$\mathcal{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{pmatrix}$$

Comme $\mathcal{A}$ et $\mathcal{B}$ ont même dimension, nous pouvons les additionner et les soustraire de la façon suivante:

$$\mathcal{A}+\mathcal{B} = \begin{pmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{m2}+b_{22} & \cdots & a_{2n}+b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \cdots & a_{mn}+b_{mn} \end{pmatrix}$$

Similairement, la différence entre deux matrices est

$$\mathcal{A}-\mathcal{B} = \begin{pmatrix} a_{11}-b_{11} & a_{12}-b_{12} & \cdots & a_{1n}-b_{1n} \\ a_{21}-b_{21} & a_{m2}-b_{22} & \cdots & a_{2n}-b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}-b_{m1} & a_{m2}-b_{m2} & \cdots & a_{mn}-b_{mn} \end{pmatrix}$$

Si $\mathcal{C}$ est une matrice $n \times p$:

$$\mathcal{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{np} \end{pmatrix}$$

Dès lors, nous pourrons définir la multiplication de matrices. En effet, notons

$$\mathcal{D} = \mathcal{A}\mathcal{C} = \begin{pmatrix} d_{11} & d_{12} & \cdots & d_{1p} \\ d_{21} & d_{22} & \cdots & d_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ d_{m1} & d_{m2} & \cdots & d_{mp} \end{pmatrix}$$

Où

$$d_{ij} = \sum_k a_{ik} c_{kj}$$

Le produit matriciel est en effet une matrice de produits scalaires entre lignes et colonnes.

Nous aurons également besoin du produit matriciel de Hadamart, parfois aussi appelé produit matriciel de Schur, qui est défini pour deux matrices de même dimensions, comme suit:

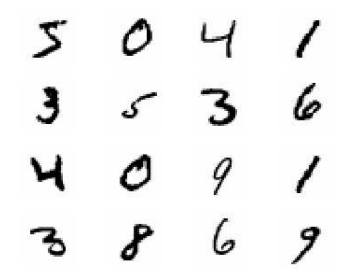
$$\mathcal{A} \circ \mathcal{B} = \begin{pmatrix} a_{11}b_{11} & a_{12}b_{12} & \cdots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{m2}b_{22} & \cdots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \cdots & a_{mn}b_{mn} \end{pmatrix}$$

Pour toute matrice, nous pouvons également définir sa transposée, qui est la symétrique de cette matrice par sa diagonale principale.

$$\mathcal{A}^T = \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1m} & a_{2m} & \cdots & a_{nm} \end{pmatrix}$$

Une notation non-universelle qu'on utilisera sera $(a_{.j})$ et $(a_{i.})$ pour désigner respectivement la colonne j et la ligne i de la matrice $\mathcal{A}$.

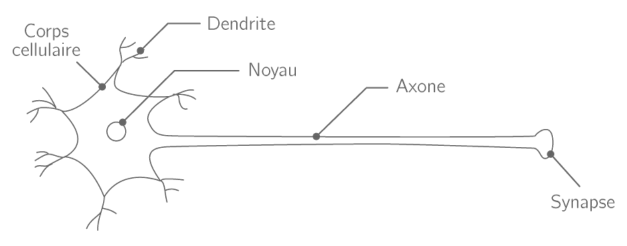
6.1 Cerveau humain



Rares sont les personnes n'étant pas capables de reconnaître cette séquence de chiffres. Nous les décodons de façon subconsciente, à l'aide de notre cerveau. Dans chaque hémisphère de celui ci se trouvent un rassemblement de plusieurs millions de neurones appelé cortex primaire de vision V1. Et ceux ci ne sont pas seuls, comme il existe V2, V3, V4 et V5, des cortex de plus en plus compliqués pour peaufiner le travail de la vision. Notre cerveau est une sorte de super-ordinateur composé de plusieurs millions voire milliards de composantes, appelés des neurones.

Pourtant cette tâche de reconnaissance de chiffres est énormément compliquée pour un ordinateur. Il est très compliqué de classifier les nombres, comme la boucle du 8 par exemple ne sera pas toujours de la même taille, ne sera pas toujours au même endroit, et plus important, elle ne sera pas toujours bien faite. Il y a donc nécessité de trouver un système ne se reposant pas sur une caractérisation précise. Pour ce faire, nous nous inspirons du système le plus performant connu aujourd'hui; le cerveau humain. Il nous faut pour ce faire comprendre de façon très vaste le fonctionnement d'un neurone.

Un neurone est une entité biologique qui assure la transmission de l'influx nerveux, étant un message électrique contenant de l'information.

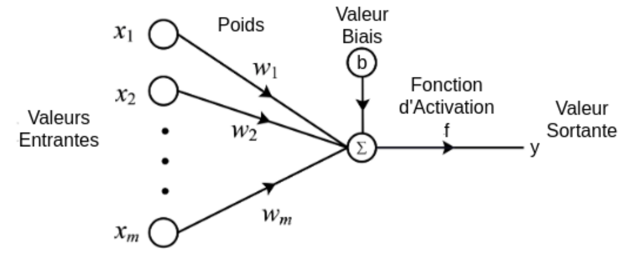


Ce dernier est composé de quelques parties. Une première partie est composée des dendrites, qui captent l'information à transmettre auprès d'autres entités. Le message reçu n'est pas forcément unique, chaque neurone peut capteur une autre information provenant d'endroits divers. Ces signaux sont alors combinés d'une façon ou d'une autre dans le corps cellulaire et est transmis à travers l'axone, qui relie les entités entre elles. Finalement, une connexion entre plusieurs entités forme une synapse, à travers laquelle peut passer le message.

6.2 Perceptron

Pour pouvoir utiliser un neurone dans notre système, il nous en faut une modélisation mathématique. On l'appellera le perceptron.

Plusieurs valeurs entrantes, représentant les signaux perçus des autres perceptrons, sont les paramètres du modèle. Ceux ci seront dès lors combinés en un signal unique. Au final, cette combinaison résultante, est transmise aux perceptrons suivants.



Comme aucun modèle n'est exact à 100%, nous nous permettons de poser une contrainte. La facon de combiner est en un premier lieu une combinaison affine, et ensuite cette valeur passera par une fonction non-affine.

Notons $(x_1, \dots, x_m)$ les valeurs entrantes d'un perceptron. Alors la valeur sortante sera, pour certains réels $w_1, \dots, w_m, b$,

$$y = f(w_1 x_1 + \cdots + w_m x_m + b) = f\big(\sum w_i x_i + b\big)$$

Les nombres réels $w_1, \dots, w_m$ sont appelés les *poids* attribués au perceptron, le nombre réel b est appelé le *biais* de celui ci, et la fonction non-linéaire f est appelée sa *fonction d'activation*.

Pour simplifier cette notation on peut noter

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix}$$

la matrice des valeurs entrantes. On notera également

$$w = \begin{pmatrix} w_1 & w_2 & \cdots & w_m \end{pmatrix}$$

la matrice des poids du système.

Dès lors, on a

$$y = f(wx + b)$$

Le perceptron forme un modèle mathématique, c'est à dire qu'il exprime une relation, représentant celle présente dans un certain système, entre une multitude de valeurs. En l'occurrence, cette relation ressemblera à

$$(x_1, \dots, x_m) \mapsto f(w_1 x_1 + \cdots w_m x_m + b)$$

Intuitivement, cette relation n'est pas affine comme f n'est pas affine. Montrons cela:

Propriété: La relation exprimée par un perceptron n'est pas une relation affine.

Preuve: Supposons que cela ne soit pas le cas. Cela veut dire que, pour tout $x_1, \dots, x_m$, on a

$$f(w_1x_1 + \cdots + w_mx_m + b) = a_0 + a_1x_1 + \cdots + a_mx_m$$

pour certains $a_0, \dots, a_m \in \mathbb{R}$.
Dès lors cela est vrai pour, x étant un paramètre réel,

$$(x_1, x_2, \dots, x_m) = \big(\frac{x-b}{w_1}, 0, \dots, 0\big)$$

Cela implique que

$$f(x) = a_0 + \frac{a_1}{w_1}(x-b)$$

Or comme ceci est vrai pour tout $x \in \mathbb{R}$, f doit être une application affine, ce qui est absurde!

Dès lors, notre supposition était absurde et la thèse démontrée. □

Maintenant que l'on a introduit le concept d'un perceptron, prenons un exemple. Notre perceptron va calculer si oui ou non une certaine personne aurait envie de financer un projet. Ce perceptron possède 3 valeurs entrantes valant soit 0 soit 1; Si la personne a compris le projet ou non, si le projet était intéressant d'un point de vue écologique et si les présentateurs étaient bien habillés. Ces valeurs sont ici supposées indépendantes. Le perceptron va alors faire de la magie et va calculer un nombre, valant soit 0, soit 1, indiquant si l'on veut voter pour ce projet ou non.

On prendra comme fonction d'activation

$$f(x) = \begin{cases} 1 & \text{Si } x \geq 0 \\ 0 & \text{Sinon} \end{cases}$$

Cette fonction s'appelle la fonction "Marche", parce que son graphe ressemble à une marche d'escalier. Il est l'une des fonctions couramment utilisées dans les perceptrons. Une liste de plusieurs fonctions d'activations sera donnée ultérieurement.

Imaginons que la personne soit raisonnable, et que l'habit des présentateurs ne l'importe que très peu, et qu'il soit assez apte à financier tout projet tant qu'il est en rapport avec l'écologie, qu'il le comprenne ou non. Les poids du perceptron seront donc dans le genre

$$w_1 = 2, w_2 = 5, w_3 = 1$$

Il faudra pourtant pas uniquement que cette somme pondérée soit positive pour gagner son financement, il faut ressortir de l'ordinaire, donc la somme doit être supérieure à 6;

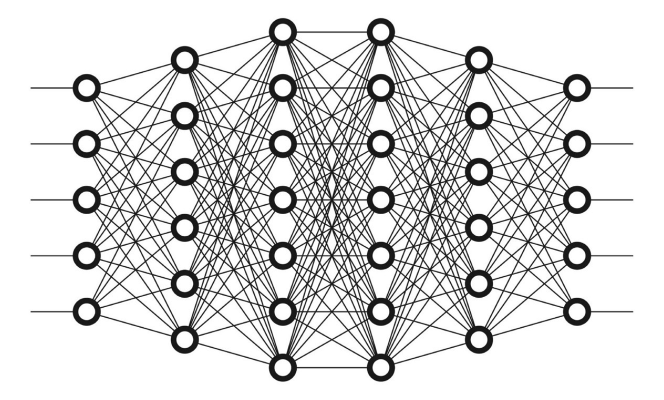
$$b = -6$$

Dès lors, un projet compliqué en lien avec l'écologie présenté par des personne bien habillées aura une valeur sortante de

$$f(1 \cdot 0 + 5 \cdot 1 + 1 \cdot 1 - 6) = f(0) = 1$$
$$\implies y = 1$$

Donc la personne en question va financer ce projet.

Le cerveau humain n'est pas composé d'un seul neurone, mais bien d'une grande pluralité combinés dans tous les sens. C'est également comme ceci que nous allons construire notre cerveau artificiel. Aucune décision n'est réellement prise comme l'exemple décrit plus haut, mais de façon bien plus complexe, c'est à dire en se servant de plus de perceptrons. Nous allons assembler ceux ci couches par couches, tels que ceux de deux couches adjacentes soient connectés entre elles, et de telle façon que deux couches voisines soient totalement connectés.



Lors de notre définition du perceptron, nous avons dit qu'un perceptron avait une unique valeur sortante. Pourtant, dans le schéma du réseau de neurones ci dessus on observe plusieurs liaisons sortantes de chaque perceptron. Ceci n'est pas une contradiction, toutes les valeurs sortantes d'un même neurone sont simplement les mêmes, représentées plusieurs fois.

Nous allons poser une contrainte pour que le système soit facile à gérer; tous les perceptrons d'une même couche sont assimilés à la même fonction activatrice.

La première couche correspond à la "couche entrante", donc chaque noeud aura la valeur d'un des paramètres du système.
La dernière couche correspond à la "couche sortante", leurs valeurs seront les résultats du modèle.
Toutes les couches entre celles ci sont des "couches cachées", qui servent pour faire des calculs intermédiaires mais dont nous ne connaîtrons jamais la réelle utilité.

6.3 Complétude fonctionnelle

Cette sous-section est un plus à l'idée du Deep Learning, qui permet de mieux comprendre son fonctionnement mais qui n'est pas nécessaire à sa compréhension. Elle permet de se construire une

intuition du fait qu'un réseau de perceptrons permet de représenter toute fonction, à condition de bien choisir sa fonction d'activation. Nous n'allons pas montrer ce résultat, mais nous allons plutôt démontrer que toute fonction logique peut être obtenue en choisissant uniquement des fonctions "marche". Cette preuve ne se généralise pas, mais elle permet de se construire une intuition qu'un système peut s'adapter à la plupart des scénarios.

Tout d'abord, il nous faut définir une fonction logique. Nous appellerons une fonction logique une fonction envoyant un k-uplet de chiffres binaires (de 0 ou de 1) sur un chiffre binaire. Nous pouvons écrire

$$g\colon \{0,1\}^k \mapsto \{0,1\}$$

Plus généralement, il n'est pas nécessaire de se limiter à 0 et à 1, nous allons plutôt travailler sur un ensemble binaire $\mathbb{B}$, étant un ensemble à 2 éléments distincts. Dès lors, on définira une fonction logique g telle que

$$g\colon \mathbb{B}^k \mapsto \mathbb{B}$$

Puisque l'ensemble domaine et d'arrivée sont finis, le nombre de telles fonctions est fini. Nous pouvons calculer leur nombre:

$$\#\{g \mid g\colon \mathbb{B}^k \mapsto \mathbb{B}\} = (\#\mathbb{B})^{\#\mathbb{B}^k} = 2^{(2^k)}$$

Pour $k = 2$, nous avons donc 16 fonctions logiques différentes. Si l'on arrive à expliciter 16 fonctions logiques distinctes, nous saurons que nous avons trouvé toutes les fonctions logiques pour $k = 2$ existantes. Faisons donc cela.

Commencons par définir une fonction de base, appelée la fonction NAND, que l'on notera $\uparrow$. Cette fonction sera définie comme suit:

p	q	$p \uparrow q$
0	0	1
1	0	1
0	1	1
1	1	0

On observe que $p \uparrow q$ vaut 0 si et seulement p et q sont tous deux 1.

Cette fonction constitue la première des 16 recherchées. Explicitions maintenant les 15 autres.

Tout d'abord, on définira une fonction, appelée la "négation", pour que les fonctions par après soient plus lisibles:

$$\neg p = p \uparrow p$$

Son nom vient du fait que

$$\neg p = \mathbb{B} \backslash \{p\}$$

Définissons maintenant les 16 fonctions recherchées:

$$\begin{aligned}
f_1(p,q) &\equiv \neg(p \uparrow \neg p)\\
f_2(p,q) &\equiv \neg(p \uparrow q)\\
f_3(p,q) &\equiv \neg(p \uparrow \neg q)\\
f_4(p,q) &\equiv p\\
f_5(p,q) &\equiv \neg(\neg p \uparrow q)\\
f_6(p,q) &\equiv q\\
f_7(p,q) &\equiv (p \uparrow \neg q) \uparrow (\neg p \uparrow q)\\
f_8(p,q) &\equiv \neg p \uparrow \neg q\\
f_9(p,q) &\equiv \neg\big(\neg p \uparrow \neg q\big)\\
f_{10}(p,q) &\equiv \neg\big((p \uparrow \neg q) \uparrow (\neg p \uparrow q)\big)\\
f_{11}(p,q) &\equiv \neg q\\
f_{12}(p,q) &\equiv \neg p \uparrow q\\
f_{13}(p,q) &\equiv \neg p\\
f_{14}(p,q) &\equiv p \uparrow \neg q\\
f_{15}(p,q) &\equiv p \uparrow q\\
f_{16}(p,q) &\equiv p \uparrow \neg p
\end{aligned}$$

Comme toute fonction est définie par rapport à $\uparrow$ et par rapport à $\neg$, qui est définie par rapport à $\uparrow$, toutes les fonctions sont définies par rapport à $\uparrow$. Elles sont également toutes distinctes. Cela peut s'observer en dressant le tableau de vérité de chacune d'entre elles et vérifier qu'il est différent pour toute fonction:

p	**0**	**0**	**1**	**1**
q	**0**	**1**	**0**	**1**
f_1	0	0	0	0
f_2	0	0	0	1
f_3	0	0	1	0
f_4	0	0	1	1
f_5	0	1	0	0
f_6	0	1	0	1
f_7	0	1	1	0
f_8	0	1	1	1
f_9	1	0	0	0
f_{10}	1	0	0	1
f_{11}	1	0	1	0
f_{12}	1	0	1	1
f_{13}	1	1	0	0
f_{14}	1	1	0	1
f_{15}	1	1	1	0
f_{16}	1	1	1	1

Dès lors les 16 fonctions logiques à 2 variables peuvent toutes être explicitées à partir de $\uparrow$.

Soit maintenant f une fonction logique quelconque à k variables. On peut décomposer toutes les opérations en fonctions logiques à 2 variables imbriquées les unes dans les autres. Les opérations doivent se dérouler dans un certain ordre, imaginons que g soit la dernière à être calculée. Dès lors,

$$f(a_1, \ldots, a_k) = g(b_1, b_2)$$

Or, comme on sait, g est une fonction logique à 2 variables donc peut être explicitée à partir de $\uparrow$. On procède de façon analogue sur b_1 et sur b_2 pour trouver que chacune d'entre elles découlent de $\uparrow$. En continuant ainsi, f est juste un enchaînement de beaucoup de $\uparrow$. Dès lors, toute fonction logique en général peut être explicitée à partir de $\uparrow$. Ceci prouve un résultat n'ayant rien à voir avec les réseaux de neurones; La fonction NAND est fonctionnellement complète. Il est intéressant de voir que les seules fonctions ayant cette propriété sont NAND et NOR, les fonctions f_{15} et f_9

Montrons maintenant que la fonction NAND peut être obtenue par un perceptron. Vu le résultat précèdent, cela implique qu'un réseau de perceptrons est fonctionnellement complet.
Prenons un perceptron à deux valeurs entrantes (Correspondant à p et q) et une valeurs sortante (Qu'on aimerait faire correspondra à $p \uparrow q$). Plusieurs poids/biais sont possibles, mais imaginons

$$w_1 = w_2 = -1, b = 1.$$

Dès lors, on observe que ce perceptron représente exactement la fonction $\uparrow$. Le perceptron peut bien entendu avoir plus de valeurs entrantes dont nous fixerons les poids à 0 pour qu'ils n'aient aucune importance.

Un autre perceptron qui sera le perceptron "égalité" aura une valeurs entrante et un poids de 1, avec une valeur biais de 0. Toutes les autres valeurs entrantes auront un poids de 0. Dès lors la valeurs sortante du perceptron sera égale à la valeurs entrante choisie du perceptron.

On peut composer toute fonction logique en un enchaînement d'opérations $\uparrow$. Dans la couche entrante de notre réseau de neurones, nous placerons tous les paramètres de la fonction logique. Sur la couche suivante, nous placerons suffisamment de perceptrons "égalité" pour pouvoir reproduire toutes les valeurs de la couche suivante, et nous placerons un perceptron $\uparrow$ pour pouvoir calculer la valeur suivante à calculer. Nous continuerons ainsi de suite jusqu'à arriver au bout de la fonction. Ceci sera en effet un très grand réseau de neurones mais il sera fini et représentera parfaitement la fonction logique choisie.

Dès lors, toute fonction logique peut être calculée par un réseau de neurones, ce qui est un résultat très puissant. Ceci ne se généralise bien entendu pas à toutes les fonctions en changeant les fonctions d'activation, comme les fonctions réelles ne sont pas dénombrables et loin d'être finies, mais on peut, grâce à ce résultat, se construire une intuition qu'un réseau de neurones permet en effet d'au moins approximer toute fonction en choisissant de bonnes fonctions d'activation.

6.4 Rétro-propagation

Reprenons le perceptron de la section 7.2 avec les mêmes notations utilisées à ce moment là, mais etendons les.
On notera

$$a^0 = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$$

la matrice des valeurs entrantes dans la couche d'entrée (Elles sont au nombre de m), et

$$w^l = \begin{pmatrix} w^l_{11} & w^l_{12} & \cdots & w^l_{1k} \\ w^l_{21} & w^l_{22} & \cdots & w^l_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ w^l_{j1} & w^l_{j2} & \cdots & w^l_{jk} \end{pmatrix}$$

la matrice des poids des perceptrons de la couche l, où w^l_{ab} est le poids de la liaison entre le $b$ième perceptron de la couche $l-1$ et le $a$ième perceptron de la couche l. La couche $l-1$ comporte un total de k perceptrons, tandis que la couche l en comporte j.

On notera aussi

$$b^l = \begin{pmatrix} b^l_1 \\ b^l_2 \\ \vdots \\ b^l_j \end{pmatrix}$$

Les valeurs biais de la couche l (Au nombre de j).
Par après, soit

$$f = (f_1, f_2, \cdots, f_L)$$

les fonctions d'activation de chacune des couches (Au nombre de L, le nombre de couches mises à part la première).
Finalement, soit

$$a^l = \begin{pmatrix} a^l_1 \\ a^l_2 \\ \vdots \\ a^l_j \end{pmatrix}$$

Les valeurs sortantes de la couche l (Au nombre de j).

On peut observer que a^0 représente les valeurs entrantes du système, donc les valeurs sortantes de la première (Ou 0ième) couche et que a^L (où L est le nombre de couches totales) sont les valeurs sortantes finales, les résultats.

Rappelons alors comment la valeur sortante est définie:

$$a^l = f_l\big(w^l a^{l-1} + b^l\big)$$

Cette forme ci est clairement plus simple et plus compréhensible que le forme de "valeur par valeur" qui ressemble à ceci:

$$a^l_i = f_l\big(\sum_j w^l_{ij} \cdot a^{l-1}_j + b^l_i\big)$$

Cette forme est pratiquement inutilisable vu sa complexité...

On va également noter

$$z^l = w^l a^{l-1} + b^l$$

On appellera z^l la somme pondérée de la couche l.

Un réseau de neurones n'est bien entendu pas adapté à ses exemples dès le début, il lui faut une phase d'apprentissage avant que cela ne soit bon. Comme le nombre de paramètre est, dans la plupart des cas, énorme, nous n'avons aucune idée de comment les initialiser pour que la phase d'apprentissage soit optimale. Nous allons donc simplement les définir de façon aléatoire.

La façon dont on va faire apprendre à notre réseau sera avec une manière qui s'est manifestée utile et efficace chez l'être humain; la méthode par essai-erreur. Il nous faut donc des exemples à montrer au système pour qu'il puisse apprendre. Pour une matrice de valeurs entrantes $\mathbf{x}$ appelons $y(\mathbf{x})$ la matrice des résultats réels (Donc pas donnés par le système mais donnés par celui supervisant le système) et $a(\mathbf{x})$ les valeurs calculées par le système.

Nous allons alors définir une fonction "erreur" $\mathbf{E}(w, b)$, qui va donner une indication d'à quel point le système était loin ou près de trouver la réponse correcte en fonction des poids et des biais du système. Elle sera parfois notée $\mathbf{E}$ pour simplifier l'écriture. Cette fonction erreur doit satisfaire plusieurs contraintes.

En un premier, cette fonction doit pouvoir s'écrire comme moyenne d'une même fonction erreur sur tous les exemples fournis, c'est à dire

$$\mathbf{E}(w, b) = \frac{1}{n} \sum_{\mathbf{x}} E_{\mathbf{x}}(w, b)$$

Avec cette supposition, on peut se contenter de ne regarder qu'une fonction erreur, disons $E_{\mathbf{x}}(w, b)$ pour une certain $\mathbf{x}$, et de faire la moyenne par après. Notons cette fonction que l'on observe E pour simplifier l'écriture par après. Se fixer une fonction revient en réalité à se fixer un $\mathbf{x}$, donc nous pouvons noter, de nouveau pour simplifier l'écriture, $y(\mathbf{x}) = \mathbf{y}$ et $a(\mathbf{x}) = \mathbf{a}$.

La seconde contrainte est que la fonction E doit dépendre de $\mathbf{y}$ et de $\mathbf{a}$, car en effet E doit donner une indice d'à quel point $\mathbf{y}$ et $\mathbf{a}$ sont proches. Nous avons pourtant notés plus haut que $E = e_{\mathbf{x}}(w, b)$ avait comme variables w et b, mais également $\mathbf{x}$. Dès lors, comme $\mathbf{y} = y(\mathbf{x})$ ne dépend que des paramètres du système et de $\mathbf{x}$, la fonction dépend bien de $\mathbf{y}$. Il faut donc que les 3 variables de E soient utiles dans son expression et que E dépende de $\mathbf{a}$ à un certain point.

La dernière contrainte est que E doit varier de façon continue lorsque w et b varient de façon continue. Cette contrainte semble sortir de nulle part pour le moment, mais elle s'avère être la plus naturelle par après.

Ce raisonnement ci est théorique, et nous ne prendrons donc aucune fonction erreur comme exemple pour le moment. Un bon nombre d'exemples seront donnés dans une section antérieure, et il est utile d'aller la voir maintenant pour se fonder une idée de ce que peut être une fonction erreur.

Comme $\mathbf{x}$ est fixé, les seules valeurs que nous pouvons changer dans le système sont les poids et les valeurs biais. En effet, nous essayons de les adapter pour que le système serve d'approximation pour le résultat voulu. Il est souvent possible de changer n'importe quel poids ou valeurs biais d'un nombre exact pour avoir le résultat exact. Cela devrait nous satisfaire, comme nous avons obtenu le bon résultat. Le problème est que lorsque nous itérons sur l'exemple suivant, nous allons de nouveau dramatiquement changer le système, et nous perdrons donc tout notre avancement obtenu après l'analyse du premier exemple. Il est donc plus prudent de changer un peu le système pour chaque exemple et de revenir sur chaque exemple plusieurs fois. Nous ne voulons pas non plus nous fixer sur seulement quelques paramètres (poids et valeurs biais), mais nous voulons changer un peu tous les paramètres. Pour cela, il nous faut évaluer ce qu'un changement à chaque paramètre influence le résultat. Comme notre but est de minimiser l'erreur, nous allons nous intéresser à ce qu'un changement d'un paramètre influence sur l'erreur, et ainsi faire varier ce paramètre un peu dans le sens voulu.

Calculer l'influence sur l'erreur d'un petit changement sur un paramètre nous rappelle quelque chose que l'on connaît déjà, notamment la dérivée. En effet, il nous suffit de calculer

$$\frac{\partial E}{\partial b_i^l} \quad \text{et} \quad \frac{\partial E}{\partial w_{ij}^l}$$

Il est bien gentil de devoir calculer ces expressions, mais un ordinateur, ne sachant pas faire du calcul symbolique sans aide, n'en est absolument pas capable. C'est pourquoi il nous faut transformer ces expressions dans des formules utilisables par un ordinateur.
Comme nous fournissons à l'ordinateur les expressions closes de la fonction erreur et des fonctions d'activation, nous admettons leurs dérivées connues également. Il suffira simplement de les calculer de main et de les fournir à l'ordinateur une fois pour toutes avant le lancement du processus d'apprentissage.

Nous allons définir l'erreur du neurone i de la couche l comme suit

$$\delta_i^l = \frac{\partial E}{\partial z_i^l}$$

Comme d'habitude, δ^l est la matrice des erreurs des neurones dans la couche l.
Pour $l = L$, on peut renoter cette équation comme suit:

$$\delta_i^L = \frac{\partial E}{\partial z_i^L} = \sum_k \frac{\partial E}{\partial a_k^L} \cdot \frac{\partial a_k^L}{\partial z_i^L}$$

Or, le second facteur du produit s'avère simple à calculer:

$$\frac{\partial a_k^L}{\partial z_i^L} = \frac{\partial f_L(z_k^L)}{\partial z_i^L}$$

Cette expression vaut 0 lorsque $k \neq i$, comme $f_L(z_k^L)$ ne dépend pas de z_i^L, et vaut

$$\frac{\partial f_L(z_i^L)}{\partial z_i^L} = f_L'(z_i^L)$$

$$\implies \delta_i^L = \frac{\partial E}{\partial a_i^L} \cdot f'(z_i^L)$$

Tout ceci est facilement calculable, comme z^L est calculé pour trouver le résultat du système, et que f est connue, donc la dérivée de f en z_i^L n'est pas spécialement compliquée. Il en va de même pour $\frac{\partial E}{\partial a_i^L}$; comme, par définition; E dépend de $a^L = \mathbf{a}$ et que sa formule est connue, donc se dérivée est évaluable.

Dès lors on a trouvé une expression matricielle pour δ^L:

$$\delta^L = \nabla_{\mathbf{a}} E \circ f'(z^L)$$

Cette équation sera la première des équations fondamentales de la rétro-propagation.

La seconde servira à évaluer δ^l pour toute couche l:

$$\delta_i^l = \frac{\partial E}{\partial z_i^l} = \sum_j \frac{\partial E}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{\partial z_i^l}$$

Or, par définition de δ,

$$\frac{\partial E}{\partial z_j^{l+1}} = \delta_j^{l+1}$$

Aussi,

$$\frac{\partial z_j^{l+1}}{\partial z_i^l} = \frac{\partial \sum_k w_{jk}^{l+1} f_l(z_k^l) + b_j^{l+1}}{\partial z_i^l}$$

$$= \sum_k \frac{\partial w_{jk}^{l+1} f_l(z_k^l) + b_j^{l+1}}{\partial z_i^l} = \frac{\partial w_{ji}^{l+1} f_l(z_i^l) + b_j^{l+1}}{\partial z_i^l}$$

$$= w_{ji}^{l+1} f_l'(z_i^l)$$

De ce fait,

$$\delta_i^l = \sum_j \delta_j^{l+1} \cdot w_{ji}^{l+1} f_l'(z_i^l) = f_l'(z_i^l) \sum_j \delta_j^{l+1} \cdot w_{ji}^{l+1}$$

$$= f_l'(z_i^l) \cdot \left((w_{i.}^{l+1})^T \delta^{l+1}\right)$$

$$\implies \delta^l = f_l'(z^l) \circ (w^{l+1})^T \delta^{l+1}$$

Cette seconde équation fondamentale permet, avec la première, de comprendre l'idée derrière la rétro-propagation. En effet, la première nous a permis de savoir ce que l'erreur de la dernière couche vaut. La seconde formule permet de savoir ce que valent toutes les autres par récurrence (Dans le sens où connaître l'erreur de la couche l nous permet de calculer celle de la couche $l-1$). Nous savons dès lors à quel point chaque neurone influence l'erreur finale. Ceci nous sert d'indicateur pour déterminer ce qu'il faut changer dans le système. Les deux équations suivantes vont nous permettre de traduire cette information en quelque chose d'utilisable, c'est à dire qu'elles nous permettront de déterminer le taux qu'influence chaque paramètre sur le résultat final.

Pour les biais, nous pouvons déterminer cette formule ci:

$$\begin{aligned}
\frac{\partial E}{\partial b_i^l} &= \sum_k \frac{\partial E}{\partial z_k^l} \cdot \frac{\partial z_k^l}{\partial b_i^l} \\
&= \sum_k \delta_k^l \cdot \frac{\partial \left(\left(\sum_j w_{kj}^l \cdot a_j^{l-1}\right) + b_k^l\right)}{\partial b_i^l} \\
&= \sum_k \delta_k^l \cdot \left(\sum_j \frac{\partial w_{kj}^l \cdot a_j^{l-1}}{\partial b_i^l} + \frac{\partial b_k^l}{\partial b_i^l}\right) \\
&= \sum_k \delta_k^l \cdot \left(\sum_j 0 + \frac{\partial b_k^l}{\partial b_i^l}\right) \\
&= \sum_k \delta_k^l \cdot \frac{\partial b_k^l}{\partial b_i^l} \\
&= \delta_i^l
\end{aligned}$$

Et finalement, pour les poids, nous pouvons écrire

$$\begin{aligned}
\frac{\partial E}{\partial w_{ij}^l} &= \sum_k \frac{\partial E}{\partial z_k^l} \cdot \frac{\partial z_k^l}{\partial w_{ij}^l} \\
&= \sum_k \delta_k^l \cdot \frac{\partial \left(\left(\sum_r w_{kr}^l \cdot a_r^{l-1}\right) + b_k^l\right)}{\partial w_{ij}^l} \\
&= \sum_k \delta_k^l \cdot \left(\sum_r \frac{\partial w_{kr}^l \cdot a_r^{l-1}}{\partial w_{ij}^l} + \frac{\partial b_k^l}{\partial w_{ij}^l}\right) \\
&= \sum_k \delta_k^l \cdot \left(\sum_r \frac{\partial w_{kr}^l \cdot a_r^{l-1}}{\partial w_{ij}^l} + 0\right) \\
&= \sum_k \sum_r \delta_k^l \cdot a_r^{l-1} \cdot \frac{\partial w_{kr}^l}{\partial w_{ij}^l} \\
&= \delta_i^l \cdot a_j^{l-1}
\end{aligned}$$

Nos quatre équations fondamentales sont donc

$$\boxed{\begin{aligned} \delta^L &= \nabla_{\mathbf{a}} E \circ f'(z^L) && (1)\\ \delta^l &= f_l'(z^l) \circ (w^{l+1})^T \delta^{l+1} && (2)\\ \frac{\partial E}{\partial b_i^l} &= \delta_i^l && (3)\\ \frac{\partial E}{\partial w_{ij}^l} &= \delta_i^l \cdot a_j^{l-1} && (4) \end{aligned}}$$

Ces expressions sont facilement calculables par un ordinateur, le produit matriciel et le produit de Hamart faisant partie de la plupart des langages de programmation modernes et étant aisément et rapidement exécutables sur un GPU. Nous pouvons donc évaluer, grâce aux formules (3) et (4), qui nécessitent les résultats intermédiaires (1) et (2), ce que chaque paramètre change au système. Les autres valeurs, notamment $\mathbf{a}, a$ et z sont calculés lors du calcul des valeurs sortantes du système, et il suffit donc de les garder en mémoire vive pour pouvoir les ré-utiliser par après.

Plus tôt nous nous sommes fixés un exemple $\mathbf{x}$. Prenons maintenant la fonction erreur $\mathbf{E}$ comme étant la moyenne de toutes les erreurs sur tous les exemples fournis. Nous savons que, pour tout paramètre α,

$$\frac{\partial \mathbf{E}}{\partial \alpha} = \frac{1}{n} \sum_{\mathbf{x}} \frac{\partial E_{\mathbf{x}}}{\partial \alpha}$$

Et ainsi, comme $\frac{\partial E_{\mathbf{x}}}{\partial \alpha}$ est aisément évaluable par les équations précédentes, $\frac{\partial \mathbf{E}}{\partial \alpha}$ l'est aussi.

Nous allons donc, après avoir évalué $\frac{\partial \mathbf{E}}{\partial \alpha}$, donc après avoir appliqué le principe de la rétro-propagation sur tous les exemples, changer tous les paramètres un peu pour les rapprocher des solutions fournies. Nous allons les changer avec un facteur γ appelé "facteur d'apprentissage". Ce facteur sera choisi par rapport au problème, mais sera généralement assez petit (de l'ordre de $0,01$). Nous changerons donc les paramètres de cette façon:

$$b_i^l \rightarrow b_i^l - \gamma \cdot \frac{\partial E}{\partial b_i^l} = b_i^l - \gamma \cdot \delta_i^l$$

$$\implies b^l \rightarrow b^l - \gamma \cdot \delta^l$$

$$w_{ij}^l \rightarrow w_{ij}^l - \gamma \cdot \frac{\partial E}{\partial w_{ij}^l} = w_{ij}^l - \gamma \cdot \delta_i^l \cdot a_j^{l-1}$$

où $x \rightarrow y$ veut dire "changer x en y".

En répétant ce processus plusieurs fois, énormément de fois si nécessaire, le système va apprendre et s'adapter de plus en plus aux exemples. Il faut bien entendu faire attention lors du choix de γ, comme une valeur disproportionnée pourrait causer une convergence trop lente pour être utile (γ trop petit) ou causer une divergence par rapport au résultat voulu (γ trop grand).

Il faut malgré tout faire attention à ne pas itérer ce processus trop de fois non-plus, comme le système vient graduellement se coller aux exemples. Si le système se colle trop aux données fournies, il s'adaptera mal à de nouvelles données comme il n'aura au final que effectué une régression de façon compliquée. Lorsqu'un système est sur-entraîné, nous parlerons de "Overfitting". Pour vérifier si l'on a trop entraîné un système, nous conservons des données exemple sur lesquelles le système ne s'est pas exercé. Nous calculons l'erreur sur ces données étant nouvelles au système et nous observons à quel point il est bon. Si le système est trop loin des données de vérification, nous n'avons soit pas assez entraîné le modèle, soit il a été trop entraîné.

6.5 Fonctions d'activations

Maintenant que nous avons proprement défini ce qu'est un réseau de perceptrons, nous allons montrer pourquoi la fonction activatrice ne peut être affine.

Propriété: Une couche dont la fonction activatrice est affine peut facilement être supprimée.

Preuve: Nous reprenons les mêmes notations que dans la partie précédente.
Supposons f_l affine pour une certaine couche l. Cela implique que, pour certains $r, t \in \mathbb{R}$, on ait

$$f(x) = rx + t$$

Appelons T une matrice de dimension $1 \times k$, où k est le nombre de perceptrons dans la couche l, remplie de valeurs t.
Dès lors

$$\begin{aligned} z^{l+1} &= w^{l+1} a^l + b^{l+1}\\ &= w^{l+1} f_l(z^l) + b^{l+1}\\ &= w^{l+1}(rz^l + T) + b^{l+1}\\ &= (rw^{l+1} z^l + w^{l+1} T) + b^{l+1}\\ &= (rw^{l+1}(w^l a^{l-1} + b^l) + w^{l+1} T) + b^{l+1}\\ &= rw^{l+1} w^l a^{l-1} + rw^{l+1} b^l + w^{l+1} T + b^{l+1} \end{aligned}$$

On peut vérifier que $rw^{l+1}w^la^{l-1}, rw^{l+1}b^l, w^{l+1}T, b^{l+1}$ ont toutes la même dimension. On effacera donc la couche l et l'on remplacer changera les paramètres en

$$w'^l = rw^{l+1} w^l$$

$$b'^l = rw^{l+1} b^l + w^{l+1} T + b^{l+1}$$

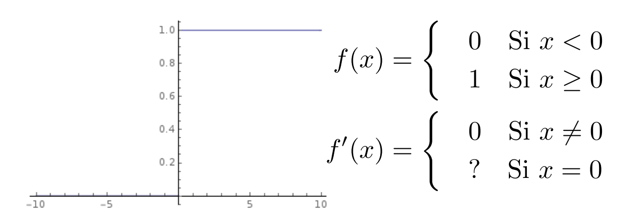
Et nous avons donc pu réduire notre réseau de une couche, ce qui aide significativement aux calculs. □

Par ailleurs, une fonction activatrice affine aurait

une dérivée constante et nous ne pourrons donc pas appliquer la rétro-propagation. Il faut donc réellement les éviter.

Comme promis, voici une liste non-exhaustive de fonctions d'activation utilisées dans les réseaux de perceptrons (Avec leur calcul de dérivée comme il est utile):

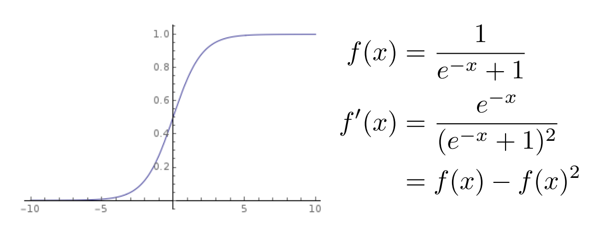
Fonction Marche:



$$f(x) = \begin{cases} 0 & \text{Si } x < 0 \\ 1 & \text{Si } x \geq 0 \end{cases}$$

$$f'(x) = \begin{cases} 0 & \text{Si } x \neq 0 \\ ? & \text{Si } x = 0 \end{cases}$$

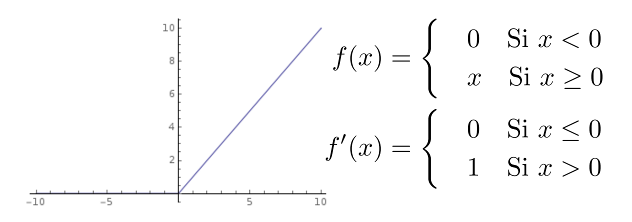
Fonction Sigmoid:



$$f(x) = \frac{1}{e^{-x} + 1}$$

$$f'(x) = \frac{e^{-x}}{(e^{-x} + 1)^2} = f(x) - f(x)^2$$

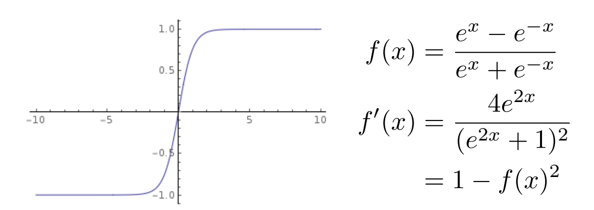
Fonction ReLU:



$$f(x) = \begin{cases} 0 & \text{Si } x < 0 \\ x & \text{Si } x \geq 0 \end{cases}$$

$$f'(x) = \begin{cases} 0 & \text{Si } x \leq 0 \\ 1 & \text{Si } x > 0 \end{cases}$$

Fonction Tangente Hyperbolique:



$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$f'(x) = \frac{4e^{2x}}{(e^{2x} + 1)^2} = 1 - f(x)^2$$

6.6 Fonctions erreur

Nous fournissons également quelques exemples de fonctions erreur (Avec leur calcul de gradient par rapport à **a** comme il est nécessaire dans les calculs):

Erreur Quadratrique

$$E_{\mathbf{x}} = \frac{1}{2} \sum_j (\mathbf{a}_j - \mathbf{y}_j)^2$$

$$\nabla_{\mathbf{a}} E = \mathbf{a} - \mathbf{y}$$

Erreur de Bernouilli

$$E_{\mathbf{x}} = \sum_j \left(\mathbf{y}_j \ln \mathbf{a}_j + (1 - \mathbf{y}_j) \ln(1 - \mathbf{a}_j)\right)$$

$$\nabla_{\mathbf{a}} E = \frac{\mathbf{a} - \mathbf{y}}{\mathbf{a} - \mathbf{a}^2}$$

Erreur Exponentielle
(τ est un paramètre)

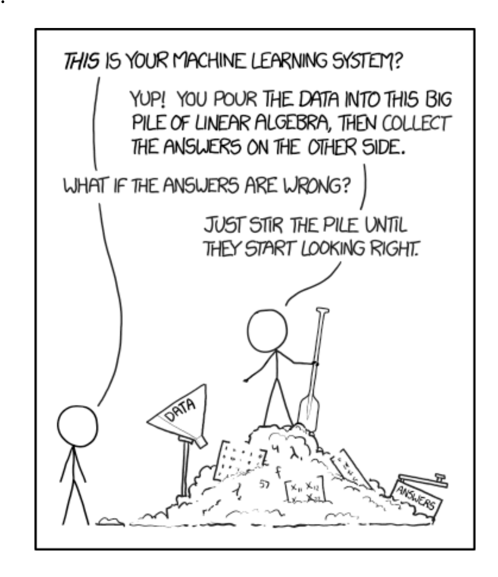
$$E_{\mathbf{x}} = \tau e^{\frac{1}{\tau} \sum_j (\mathbf{a}_j - \mathbf{y}_j)^2}$$

$$\nabla_{\mathbf{a}} E = \frac{2}{\tau} (\mathbf{a} - \mathbf{y}) E$$

6.7 Problème majeure

Les indices intermédiaires calculés par le système sont impossibles à déterminer, ils sont trop complexes et clairement incompréhensibles par l'humain. Cela est le problème majeure des réseaux de perceptrons, on ne sait réellement comment ni pourquoi ils arrivent au bon résultat. Le système ne peut être interprété et ainsi le travail ne peut être fait de cette façon par un humain. Dans des cas où le résultat est très important, cela pose un problème de faire aveuglement confiance à l'ordinateur sans pouvoir comprendre le cheminement de ses idées. Des intelligences artificielles faites pour imiter un juge lors d'un procès ont déjà été créés. L'ordinateur décide dès lors, sans pouvoir donner son raisonnement, si une personne est coupable ou non. Ceci est problématique, comme tout décision sera forcément contestée.

Dans le cas d'un jeu, il nous est pas vital de savoir comment la machine a gagné, tant qu'elle a gagné. Dans des cas plus complexes, un agent Deep Learning est souvent présent pour assister le choix final, sans pour autant le prendre tout seul. Ceci sert à faire accepter le choix par le public et a éviter des erreurs bêtes.

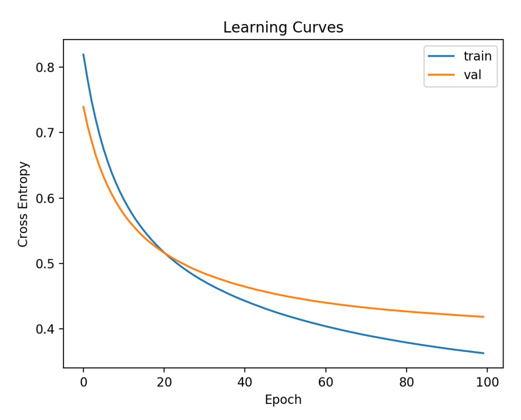


6.8 Un Exemple

Il est difficile d'illustre le Deep Learning sur papier, comme c'est plus un algorithme à faire fonctionner à travers un ordinateur. Nous avons fait fonctionner cet algorithme sur un ordinateur, et nous vous fournissons quelques résultats.

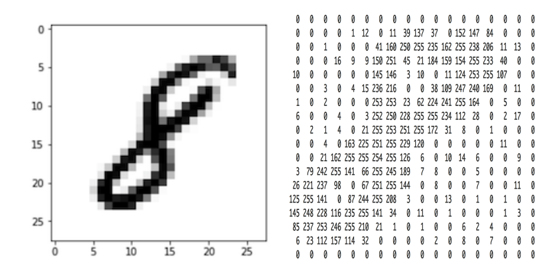
Le problème que nous envisageons est un problème de reconnaissance de chiffres. En effet, une grande base de données avec des chiffres écrits à la main sur une grille 28 × 28 avec la réelle valeur écrite dessus est fournie par MNIST. Notre tâche est d'utiliser cette base de données comme exemples pour entraîner un modèle capable de reconnaître des chiffres écrits à la main par une personne quelconque.

Nous allons utiliser une erreur de Bernouilli, également appelée "Cross Entropy". Une partie des données exemples seront isolée et utilisées comme données de validation (Pour vérifier si notre modèle s'est bien adapté). Nous savons que l'erreur sur les données exemples va converger vers 0 si le nombre d'épisodes est suffisamment grand, mais ce qu'on veut c'est que l'erreur sur les données de validation le font également. Nous savons que cette dernière ne peut en aucun cas converger aussi vite que celle de l'erreur des données exemple par contre. Nous aimerions bien que les graphes de ces deux erreurs ressemblent au suivant:

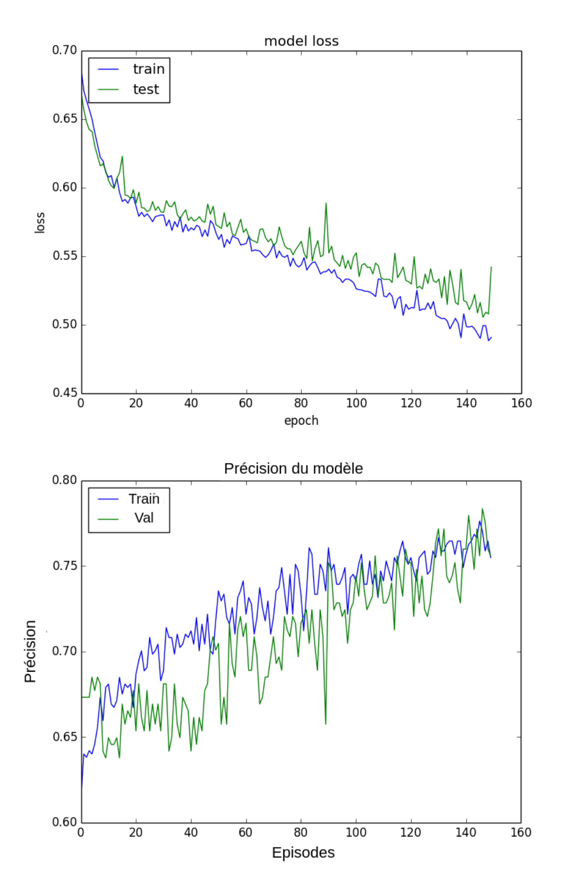


Si le graphe ressemble réellement à cela, il suffit de faire un nombre suffisant d'épisodes pour voir l'erreur de validation tendre vers 0. Bien entendu, nous arrivons au problème de "overfitting" déjà expliqué, ce qui limite le nombre d'épisodes qu'on peut faire. Si l'erreur de validation n'est pas suffisamment proche de 0 à un moment quelconque, nous avons peut-être mal choisi la fonction erreur, ou mal initialisé le système (Nombre de couches, taille des couches, fonctions d'activation des couches). Il est aussi possible que les données n'aies pas de lien entre elles et que l'intelligence n'est pas capable de prédire les données suivantes.

Dans notre cas, nous savons que l'intelligence est capable de lire des chiffres écrits à la main. Nous avons pris un réseau de perceptrons composé de 3 couches cachées, comportant respectivement 200, 100 et 50 perceptrons.
La couche d'entrée comporte $28 \cdot 28 = 784$ perceptrons, un pour chaque case de la grille. La valeur du perceptron sera la valeur de la case, valant un nombre entre 0 et 255, dépendant si case est blanche ou noire ou entre les deux.
La couche sortant contient 10 perceptrons, un pour chaque chiffre. Chaque perceptron aura une valeur sortante, représentant une probabilité que le chiffre initial soit celui lui étant accordé.



Après plusieurs épisodes, nous avons fait des graphiques avec les erreurs exemples et de validation, et de la précision du système. La précision du système est un pourcentage donnant le nombre de fois que son perceptron sortant avec la plus grande valeur était réellement la bonne réponse. C'est donc un indicateur du nombre de fois que le système avait raison.



La graphique de l'erreur des données de validation suit environ la courbe espérée, ce qui est très bon signe. Il faudra bien entendu encore faire des entraînements, comme nous n'atteignons pas encore les 80%, mais il n'est pas possible d'atteindre beaucoup plus avec ce modèle. En effet, il faudrait introduire le concept de convolution pour pouvoir définitivement passer le cap de 80% et pouvoir même atteindre les 90%.

7 Deep Q-Learning

Comme son nom l'indique, le Deep Q-Learning combine le Deep Learning (Section 6) et le Q-Learning (Section 5). Cette combinaison sera la combinaison qui pourra tuer tout (Ou presque tout) jeu. En effet, le Deep Learning ne s'adapte pas bien aux jeux, comme il faut énormément d'exemples. Le Q-Learning n'avait pas le problème d'exemples, mais il devait parcourir toutes les possibilités (Tout le tableau Q). Si l'on combine, et que l'on laisse le Deep Learning compléter le tableau Q, nous trouverons un algorithme très puissant capable de s'approprier un jeu en un rien de temps.

Rappelons la formule du Q-Learning:

$$Q(S_t, A_t) = Q(S_t, A_t) + \\ + \alpha \cdot [R_{t+1} + \gamma \cdot \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

Ce qui se réécrit comme suit:

$$Q(S_t, A_t) = (1-\alpha)Q(S_t, A_t) + \\ + \alpha \cdot R_{t+1} + \alpha \cdot \gamma \cdot \max_a Q(S_{t+1}, a)$$

Nous allons l'adapter aux jeux d'échecs. Nous ne rappelons pas les règles ici comme c'est un jeu universellement connu et facilement retrouvable ailleurs. Nous allons fixer un certain système de récompenses:
• Si le joueur prend un pion à ce tour, il a une récompense de 1.
• Si le joueur prend une pièce n'étant pas un pion à ce tour, il a une récompense de 2 • Si le joueur prend rien à ce tour, il a une récompense de -1.
• Si le joueur met l'autre joueur en échecs et mat, il a une récompense de 10. • Si le joueur peut être mis en échec et mat au tour suivant, il a une récompense de -10.

Ce système est bien entendu très naïf, comme on sait tous que prendre la reine devrait valoir plus que de prendre une tour. Il y a toujours moyen de peaufiner son système de récompenses, et on vous laisse faire comme cela vous plaît.

L'idée du Deep Q-Learning sera de créer un réseau de neurones capable de calculer les valeurs du tableau Q. Il sera toujours initialisé avec des valeurs aléatoires.

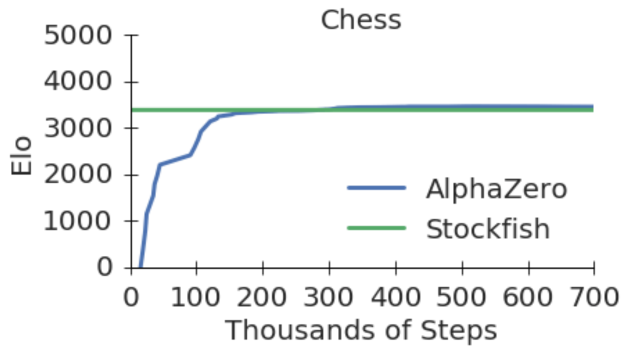
Nous allons jouer des jeux jeu par jeu. Après chaque jeu, nous allons calculer les différentes valeurs du tableau Q, comme dans le Q-Learning. Pourtant, nous n'aurons pas du tout un tableau complet, et chaque valeur ne peut être déterminée par d'autres valeurs du tableau. Au lieu d'utiliser le tableau Q pour déterminer $\max_a Q(S_{t+1}, a)$, nous allons utiliser le système pour en déterminer une approximation plus ou moins correcte. Ensuite, nous allons entraîner le système neuronal sur ces valeurs. Bien qu'elles ne soient pas complètes, et sûrement pas correctes, elles seront toujours plus correctes que celles données par le réseau, et aideront donc le réseau à s'approcher des valeurs correctes. Après la session d'entraînement sur ce nombre d'exemples très limité, nous effaçons tout le tableau Q à nouveau et nous recommençons.

Si l'on appelle le modèle du réseau neuronal M, et que nous regardons le modèle comme une fonction $M(S, A)$ qui prend comme variables un état et une action, alors nous pouvons écrire la formule du Deep Q-Learning comme suit:

$$Q(S_t, A_t) = (1-\alpha)M(S_t, A_t) + \\ + \alpha \cdot R_{t+1} + \alpha \cdot \gamma \cdot \max_a M(S_{t+1}, a)$$

Nous allons par après entraîner le modèle M sur les valeurs de Q que nous ayons déterminées.

L'avantage de cette stratégie est qu'elle s'entraîne seule. Il ne faut pas un professionnel à côté pour l'entraîner pendant des années voire plus. Le système va jouer contre lui même, donc les deux joueurs commencent aléatoirement et peaufinent leur stratégie au fur et à mesure. La seule chose qu'il faut pour pouvoir créer un tel système est un ordinateur suffisamment puissant doté d'un algorithme capable de calculer les récompenses à chaque état, ce qui n'est pas très compliqué.



Sur ce graphique, on peut voir qu'après seulement 300000 épisodes, l'agent AlphaZero, qui fonctionne avec une méthode similaire au Deep Q-Learning, a battu l'agent Stockfish, le meilleur ordinateur aux échecs jusqu'à là. L'agent Stockfish a battu tous les humains ayant joué contre lui, et a été nominé le meilleur joueur d'échecs au monde. Il fonctionne sur un algorithme simple (Section 2) très compliqué. Cette victoire de AlphaZero ne fait que montrer la supériorité du Deep Q-Learning sur l'enchaînement "stupide" d'étapes pré-dictées.

References

[1] Amir, R., and Evstigneev, I. V. *On Zermelo's theorem*. 2016.

[2] Blum, A. *Neural Networks in C++*. 1992.

[3] Brownlee, J. *Display Deep Learning Model Training History in Keras*. 2016.

[4] Brownlee, J. *TensorFlow 2 Tutorial: Get Started in Deep Learning With tf.keras*. 2019.

[5] Harrington, P. *Machine Learning in Action*. 2012.

[6] Huan, J. J. *Training and Implementing AlphaZero to play Hex*. 2018.

[7] Jain, R. *Convolutional neural networks*. 2017.

[8] Janusz. *A list of cost functions used in neural networks, alongside applications*. 2015.

[9] JavaTPoint. *Graph Theory: Tree and Forest*.

[10] Kevin Crowley, R. S. S. *Flexible Strategy Use in Young Children's Tic-Tac-Toe*. 1993.

[11] LeCun, Y. *The MNIST Database of handwritten digits*.

[12] LeCun, Y. *Efficient BackProp*. 1998.

[13] Lyuu, Y.-D. *Functional Completeness*. 2016.

[14] Melo, F. S. *Convergence of Q-learning: a simple proof*.

[15] Melo, F. S., and Ribeiro, M. I. *Convergence of Q-learning with linear function approximation*. 2007.

[16] MIT. *Game Trees*. 2009.

[17] Moris, N. M. *Digital Electronic Circuits and Systems*. 1974.

[18] Nielsen, M. *Neural Networks and Deep Learning*. 2015.

[19] Osama. *A Simple Chess Minimax*. 2012.

[20] Paul, S. *An introduction to Q-Learning: Reinforcement Learning*. 2019.

[21] S.Adamchik, V. *Introduction to AI Techniques: Game Search, Minimax, and Alpha Beta Pruning*. 2009.

[22] Sengupta, I. *Functional Completeness*.

[23] Simonini, T. *An introduction to Deep Q-Learning: let's play Doom*. 2018.

[24] Stroud, C. E. *Elementary Logic Gates*.

[25] Sutton, R. S., and Barto, A. G. *Reinforcement Learning: An Introduction*. 2018.

[26] Trask, A. *A Neural Network in 13 lines of Python (Part 2 - Gradient Descent)*. 2015.

[27] Xavier Glorot, A. B., and Bengio, Y. *Deep Sparse Rectifier Neural Networks*.

[28] Zurada, J. M. *Introduction to Artificial Neural Systems*. 1992.